09/981,844

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G09G — A2

(11) International Publication Number: **WO 98/18117**

(43) International Publication Date: 30 April 1998 (30.04.98)

(21) International Application Number: PCT/US97/18268

(22) International Filing Date: 7 October 1997 (07.10.97)

(30) Priority Data:
08/726,521 7 October 1996 (07.10.96) US

(71) Applicant: COGNEX CORPORATION [US/US]; One Vision Drive, Natick, MA 01760–2059 (US).

(72) Inventors: MICHAEL, David, J.; 901 Windsor Drive, Framingham, MA 01701 (US). WALLACK, Aaron, S.; 9 Penacook Lane, Natick, MA 01760 (US).

(74) Agent: POWSNER, David, J.; Choate, Hall & Stewart, Exchange Place, 53 State Street, Boston, MA 02109 (US).

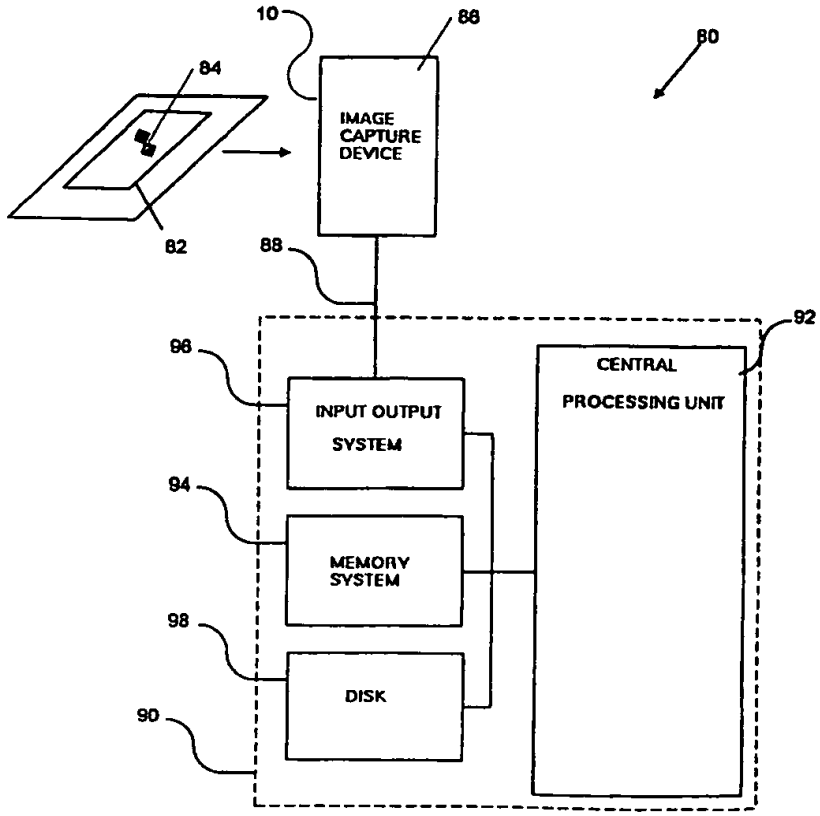
(81) Designated States: CA, JP, KR, SG, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: MACHINE VISION CALIBRATION TARGETS AND METHODS OF DETERMINING THEIR LOCATION AND ORIENTATION IN AN IMAGE

(57) Abstract

A machine vision method analyzes a calibration target of the type having two or more regions, each having a "imageable characteristic", e.g., a different color, contrast, or brightness, from its neighboring region(s). Each region has at least two edges – referred to as "adjoining edges" – that are linear and that are directed toward and, optionally meet at, a reference point (e.g., the center of the target or some other location of interest). The method includes generating an image of the target, identifying in the image features corresponding to the adjoining edges, fitting lines to those edges, and determining the orientation and/or position of the target from those lines.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

MACHINE VISION CALIBRATION TARGETS AND METHODS OF DETERMINING THEIR LOCATION AND ORIENTATION IN AN IMAGE

Reservation of Copyright

Background of the Invention

The invention pertains to machine vision and, more particularly, to calibration targets and methods for determining their location and orientation in an image.

Machine vision refers to the automated analysis of an image to determine characteristics of objects and other features shown in the image. It is often employed in automated manufacturing lines, where images of components are analyzed to determine placement and alignment prior to assembly. Machine vision is also used for quality assurance. For example, in the pharmaceutical and food packing industries, images of packages are analyzed to insure that product labels, lot numbers, "freshness" dates, and the like, are properly positioned and legible.

In many machine vision applications, it is essential that an object whose image is to be analyzed include a calibration target. Often a cross-shaped symbol, the target facilitates determining the orientation and position of the object with respect to other features in the image. It also facilitates correlating coordinate positions in the image with those in the "real world," e.g., coordinate positions of a motion stage or conveyor belt on which the object is placed. A calibration target can also be used to facilitate determining the position and orientation of the camera with respect to the real world, as well as to

facilitate determining the camera and lens parameters such as pixel size and lens distortion.

In addition to cross-shaped marks, the prior art suggests the use of arrays of dots, bulls-eyes of concentric circles, and parallel stripes as calibration targets. Many of these targets have characteristics that make difficult finding their centers and orientations. This typically results from lack of clarity when the targets and, particularly, their borders are imaged. It also results from discrepancies in conventional machine vision techniques used to analyze such images. For example, the edges of a cross-shaped target may be imprecisely defined in an image, leading a machine vision analysis system to wrongly interpret the location of those edges and, hence, to misjudge the mark's center by a fraction of a pixel or more. By way of further example, a localized defect in a camera lens may cause a circular calibration mark to appear as an oval, thereby, causing the system to misjudge the image's true aspect ratio.

In addition to the foregoing, many of the prior art calibration targets are useful only at a limited range of magnifications. Parallel stripes, for example, do not provide sufficient calibration information unless many of them appear in an image. To accommodate this, a machine vision system must utilize lower magnification. However, as the magnification decreases, so does the ability of the machine vision equipment to distinguish between individual stripes. Similar drawbacks limit the usefulness of the other prior art calibration targets for use in all but a narrow range of magnifications.

Though the art suggests the use of checkerboard patterns as alignment marks, the manner in which images of those marks are analyzed by conventional machine systems also limits their utility to a limited range of magnifications. Particularly, prior art systems obtain alignment information from checkerboard marks by identifying and checking their corners, e.g., the eight black (or white) corners in a black-and-white image. By relying on corners, the systems necessitate that images show entire checkerboards, yet, with sufficient resolution to insure accurate detection and analysis.

An object of this invention is to provide an improved calibration targets and methods for machine vision analysis thereof.

A related object is to provide calibration targets and analysis methods reliable at a wide range of magnifications.

A further object is to provide such methods as can be readily implemented on conventional digital data processors or other conventional machine vision analysis equipment.

Yet still another object of the invention is to provide such methods that can rapidly analyze images of calibration target without undue consumption of resources.

Summary of the Invention

The foregoing objects are among those attained by the invention, which provides in one aspect a machine vision method for analysis of a calibration target of the type having two or more regions, each having a different "imageable characteristic" (e.g., a different color, contrast, or brightness) from its neighboring region(s). Each region has at least two edges -- referred to as "adjoining edges" -- that are linear and that are directed toward and, optionally meet at, a reference point (e.g., the center of the target or some other location of interest). The method includes generating an image of the target, identifying in the image features corresponding to the adjoining edges, and determining the orientation and/or position of the target from those edges.

In another aspect, the invention provides a method as described above for analyzing a target of the type that includes four regions, where the adjoining edges of each region are perpendicular to one another, and in which each region in the target has a different imageable characteristic from its edge-wise neighbor. The edges of those regions can meet, for example, at the center of the target, as in the case of a four-square checkerboard.

In yet another aspect, the invention provides a method as described above for determining an orientation of the target as a function of the angle of the edges identified in the image and for determining the location of the reference point as an intersection of lines fitted to those edges. In regard to the former, the invention provides a method of determining the orientation of a target in an image by applying a Sobel edge tool to the image to generate a Sobel angle image, and by generating a angle histogram from that angle image. In an alternate embodiment, the orientation is determined by applying a Hough line tool to the image and determining the predominant angle of the edges identified by that tool.

In regard to the location of the reference point, one aspect of the invention calls for locating the adjoining edges by applying a caliper vision tool to the image, beginning at an approximate location of the reference point. That approximate location of the reference

point can itself be determined by applying a Hough line vision tool to the image in order to find lines approximating the adjoining edges and by determining an intersection of those lines. Alternatively, the approximate location of the reference point can be determined by performing a binary or grey scale correlation to find where a template representing the edges most closely matches the image.

In another alternate embodiment, the approximate location of the reference point is determined by applying a projection vision tool to the image along each of the axes with which the adjoining edges align. A first difference operator vision tool and a peak detector vision tool are applied to the output of the projection tool (i.e., to the projection) in order to find the approximate location of the edges.

The invention has wide application in industry and research applications. It facilitates the calibration of images by permitting accurate determination of target location and orientation, regardless of magnification. Thus, for example, an object bearing a target can be imaged by multiple cameras during the assembly process, with accurate determinations of location and orientation made from each such image.

These and other aspects of the invention are evident in the drawings and in the description that follows.

Brief Description of the Drawings

A more complete understanding of the invention may be attained by reference to the drawings, in which:

Figure 1A - 1C depict calibration targets according to the invention;

Figure 1D depicts the effect of rotation on the target depicted Figure 1B;

Figure 2 depicts an object according to the invention incorporating a calibration target of the type depicted in Figure 1B;

Figure 3 depicts a machine vision system according to the invention for determining the reference point and orientation of a calibration target;

Figures 4 and 5 depict a method according to the invention for interpreting an image of a calibration target to determine a reference point and orientation thereof; and

Figure 6 illustrates the magnification invariance of a target according to the invention.

Detailed Description of the Illustrated Embodiment

Figures 1A - 1C depict calibration targets according to the invention. Referring to Figure 1A, there is shown a target 10 according to the invention having three regions 12, 14, 16. Each region is bounded by at least two linear edges that are oriented toward a reference location or reference point 18 on the target. Thus, for example, region 12 is bounded by edges 20, 24; region 14 is bounded by edges 20, 22; and region 16 is bounded by edges 22, 24. As evident in the drawings, the edges are shared by adjoining regions and, hence, are referred to below as "adjoining edges." Thus, region 12 shares edge 20 with region 14; region 14 shares edge 22 with region 16; and region 16 shares edge 24 with region 12. In the illustration, the reference point 14 is at the center of target 10, though, those skilled in the art will appreciate that the reference point can be positioned elsewhere.

Each of the regions has a different imageable characteristic from its neighboring regions. As used herein, an "imageable characteristic" is a characteristic of a region as imaged by a machine vision system (e.g., of the type shown in Figure 3) and, particularly, as imaged by an image capture device used by such a system. For example, in the illustration, region 12 has the characteristic of being colored black; region 14, white; and region 16, gray. In addition to color, imageable characteristics useful with conventional machine vision systems -- which typically utilize image capture devices operational in visual spectrum -- include contrast, brightness, and stippling.

Those skilled in the art will appreciate that any other characteristics by which a region may be identified and distinguished in an image are suitable for practice of the invention. Thus, for example, for a machine vision system that utilizes a temperature-sensitive (or infrared) image capture device, an imageable characteristic is temperature. By way of further example, for a machine vision system that utilizes a nuclear decay radiation-sensitive image capture device, an imageable characteristic is emitted radiation intensity or frequency.

As shown in the illustration, the adjoining edges 20, 22, 24 comprise straight linear segments. Those edges are implicitly defined as the borders between regions that, themselves, have different imageable characteristics. Thus, for example, edge 20 is a straight linear segment defined by the border between black region 12 and white region 14. Likewise, edge 24 is defined by the border between black region 12 and gray region 16. Further, edge 22 is defined by the border between white region 14 and grey region 16.

Figure 1B depicts a calibration target 30 according to the invention having four rectangular (and, more particularly, square) regions 32, 34, 36, 38. As above, each region is bounded by at least two linear edges that are oriented toward a reference point 40 at the center of the target. Thus, for example, region 32 is bounded by edges 42, 44; region 34 is bounded by edges 42, 46; and so forth. As above, these edges are shared by adjoining regions. Thus, region 32 shares edge 42 with region 34, and so forth. Each region in target 30 has a different imageable characteristic from its edge-wise neighbor. Hence, regions 32 and 36 are white, while their edge-wise adjoining neighbors 34, 38 are black.

Figure 1C depicts a calibration target 50 according to the invention having five regions 52, 54, 56, 58, 60, each having two linear edges directed toward a reference point 62. The adjoining regions are of differing contrast, thereby, defining edges at their common borders, as illustrated. Although the edges separating the regions 52 - 60 of target 50 are directed toward the reference point 62, they do not meet at that location. As evident in Figure 1C, no marker or other element imageable characteristic is provided at reference point 62.

Those skilled in the art will appreciate that, in addition to the calibration targets shown in Figures 1A - 1C, targets with still more regions (or as few as two regions) and shapes, otherwise in accord with the teachings hereof, fall within the scope of the invention. Moreover, it will be appreciated that targets may be of any size and that their regions need not be of uniform size. Still further, it will be appreciated that the outer borders of the targets need not be linear and may, indeed, take on any shape.

Figure 2 depicts an object according to the invention for use in machine vision imaging, detection, and/or manipulation having a calibration target according to the invention coupled thereto. In the illustration, the object is an integrated circuit chip 70 having coupled to the casing thereof a calibration target 72 of the type shown in Figure 1B. Other targets according to the invention, of course, can likewise be coupled to the object 70. The targets can be coupled to the object by any known means. For example, they can be molded onto, etched into, or printed on the surface of the object. By way of further example, decals embodying the targets can be glued, screwed or otherwise affixed to the object. Moreover, by way of still further example, calibration plates incorporating the targets can be placed on the object and held in place by friction. In addition to integrated circuit chips, the object can include any other objects to which a target can be coupled, such as printed circuit boards, electrical components, mechanical parts, containers, bottles, automotive parts, paper goods, etc.

Figure 3 depicts a machine vision system 80 according to the invention for determining the reference point and orientation of an object 82 having coupled thereto a calibration target 84 according to the invention and, particularly, a four-region target of the type shown in Figure 1B. The system 80 includes an image capture device 86 that generates an image of a scene including object 82. Although the device may be responsive to the visual spectrum, e.g., a conventional video camera or scanner, it may also be responsive to emissions (or reflections) in other spectra, e.g., infrared, gamma-ray, etc. Digital image data (or pixels) generated by the capturing device 86 represent, in the conventional manner, the image intensity (e.g., contrast, color, brightness) of each point in the field of view of the capturing device.

That digital image data is transmitted from capturing device 86 via a communications path 88 to an image analysis system 90. This can be a conventional digital data processor, or a vision processing system of the type commercially available from the assignee hereof, Cognex Corporation, as programmed in accord with the teachings hereof to determine the reference point and orientation of a target image. The image analysis system 90 may have one or more central processing units 92, main memory

94, input-output system 96, and disk drive (or other mass storage device) 98, all of the conventional type.

The system 90 and, more particularly, central processing unit 92, is configured by programming instructions according to teachings hereof for operation as illustrated in Figure 4 and described below. Those skilled in the art will appreciate that, in addition to implementation on a programmable digital data processor, the methods and apparatus taught herein can be implemented in special purpose hardware.

Referring to Figure 4 there is shown a machine methodology according to the invention for interpreting an image of a target 84 to determine its reference point and orientation. The discussion that follows is particularly directed to identifying a four-region target of the type shown in Figure 1B. Those skilled in the art will appreciate that these teachings can be readily applied to finding targets according to the invention, as well as to other targets having detectable linear edges that are oriented toward a reference location or reference point on the target, e.g., a prior art cross-shaped target. For convenience, in the discussion that follows, such linear edges are referred to as "adjoining edges," regardless of whether they are from calibration targets according to the invention or from prior art calibration targets.

In step 100, an image of the target 84 (or of the target 84 and object 82) is generated, e.g., using image capture device 86, and input for machine vision analysis as discussed below. The image can be generated real time, retrieved from a storage device (such as storage device 98), or received from any other source.

In steps 102 - 108, the method estimates the orientation of the target in the image using any of many alternative strategies. For example, as shown in step 102, the method determines the orientation by applying a conventional Hough line vision tool that finds the angle of edges discernable in the image. In instances where the target occupies the entire image, those lines will necessarily correspond to the adjoining edges. Where, on the other hand, the target occupies only a portion of the image, extraneous edges (e.g., from other targets) may be evident in the output of that tool. Although those extraneous edges can

generally be ignored, in instances where they skew the results, the image can be windowed so that the Hough vision tool is only applied to that portion that contains the target. Once the angles of the lines has been determined by the Hough line tool, the orientation of the image is determined from the predominant ones of those angles. Alternatively, the angle of the image can be determined by taking a histogram of the angles.

The Hough vision tool used in step 102 may be of the conventional type known and commercially available for finding the angle of lines in image. A preferred such tool is the Cognex Line Finder, commercially available from the Assignee hereof, Cognex Corporation.

An alternative to using a Hough vision tool is shown in step 106. There, the illustrated method determines the orientation of the target by applying a Sobel edge tool to the image to find the adjoining edges. Particularly, that tool generates a Sobel angle image that reveals the direction of edges in the image. As above, where the target occupies the entire image, the adjoining edges will be the only ones discerned by the Sobel edge tool image. Where, on the other hand, the target occupies only a portion of the image, any extraneous edges can be ignored or windowed out.

The Sobel edge tool may be of the conventional type known and commercially available for finding lines in image. A preferred such tool is the Cognex Edge Detection tool, commercially available from the Assignee hereof, Cognex Corporation.

Once the Sobel angle image is generated, in step 106, the orientation of the target in the image is determined by generating a histogram of the edge angle information; see, step 108. From that histogram, the target orientation can be determined by taking a one-dimensional correlation of that histogram with respect to a template histogram of a target oriented at 0°. Where a Sobel magnitude image is generated, in addition to the Sobel angle image, such a histogram can be generated by counting the number of edges greater then a threshold length at each orientation.

As a still further alternative to applying a Hough vision tool or Sobel edge tool, the method contemplates obtaining the angle of orientation of the target from the user (or operator). To this end, the user may enter angle orientation information via a keyboard or other input device coupled with digital data processor 90.

In steps 110 - 118, the method determines the location, i.e., coordinates, of the target reference point in the image. Particularly, in step 110, the method can apply a Hough vision tool, as described above, to find the angle of lines discernable in the image. A conventional Hough vision tool determines, in addition to the angle of lines in an image, the distance of each line, e.g., from a central pixel. As above, where the target occupies the entire image, those lines will be the only ones discernable by the Sobel edge tool image. Where, on the other hand, the target occupies only a portion of the image, any extraneous edges can be ignored or windowed out.

As above, the Hough vision tool used in step 104 may be of the conventional type known and commercially available for finding the angle and position of lines in image. Once again, a preferred such tool is the Cognex Line Finder, commercially available from the Assignee hereof, Cognex Corporation. Those skilled in the art will, of course, appreciate that steps 102 and 110 can be combined, such that a single application of the Hough vision tool provides sufficient information from which to determine both the orientation of the target in the image and its reference point.

As an alternative to using a Hough vision tool, the method can apply a projection vision tool to the image in order to find the position of the lines discernable in the image; see, step 112. The projection tool, which maps the two-dimensional image of the target into a one-dimensional image, is applied along the axes defined by the edges in the image. As those skilled in the art will appreciate, the location of the edges can be discerned from by finding the peaks in the first derivatives of each of those projections. As above, where the target occupies the entire image, those lines will be the only lines discernable by the Sobel edge tool image. Where, on the other hand, the target occupies only a portion of the image, any extraneous edges can be ignored or windowed out.

The projection vision tool used in step 112 may be of the conventional type known and commercially available for mapping a two-dimensional image of the target into a one-dimensional image. A preferred such tool is that provided with the Cognex Caliper tool commercially available from the Assignee hereof, Cognex Corporation.

In step 114, the method uses the information generated in steps 110 and 112 to compute the location of the reference point, particularly, as the intersection of the lines found in those steps 110 and 112.

As an alternative to using the Hough vision tool and the projection tool, the method can apply determine the location of the reference point by performing a binary or grey scale correlation on the image; see step 116. To this end, the method uses, as a template, a pattern matching the expected arrangement of the sought-after edges, to wit, a cross-shaped pattern in the case of a target of the type shown in Figure 1B. The use of correlation vision tools for this purpose is well known in the art. The template for such an operation is preferably generated artificially, although it can be generated from prior images of similar targets.

As still another alternate to the Hough vision tool and the projection tool, the method can apply a grey-scale image registration using the sum of absolute differences metric between the image and a template; see step 118. To this end, the method uses, as a template, a pattern matching the expected arrangement of the sought-after edges, to wit, a cross-shaped pattern in the case of a target of the type shown in Figure 1B. The template for such an operation is preferably generated artificially, although it can be generated from prior images of similar targets. A preferred grey-scale image registration tool is disclosed in United States Patent No. 5,548,326, the teachings of which are incorporated herein by reference.

Although steps 110 - 118 can be used to determine the approximate location of the reference point of the target in the image, the method utilizes optional steps 120 and 122 to refine that estimate. These two steps are invoked one or more times (if at all) in order make that refinement.

In step 120, the method applies a conventional caliper vision tool to find points in the image that define the adjoining edges of the regions. On the first invocation of step 120, the method applies calipers along fifty points along each edge (though those skilled in the art will appreciate that other numbers of points can be used), beginning with points closest to the estimate of the reference point, as discerned in steps 110 - 118. The calipers are preferably applied a small distance away from the actual estimate of the reference point to avoid skewing the analysis due to possible misprinting of the target at that point, a missing pattern at that point (e.g., Figure 1C), or a too-high spatial frequency at that point (e.g., Figure 1B). In step 120, the method then fits a line to the points found along each edge by the caliper tool, preferably, using a conventional least squares technique.

In step 122, the method computes a refined location of the reference point as the intersection of the lines identified in step 120. Although conventional geometric calculations can be performed for this purpose, preferably, the reference point is computed using conventional least squares techniques. Preferably, when examining the image of a symmetric calibration target of the type shown in Figure 1B, the method utilizes the same number of points on either side of (and closest to) the reference point for purposes of fitting each line. This minimizes the bias otherwise introduced by a conventional edge detection technique in finding edges that are defined only by dark-to-light (or light-to-dark) transitions.

Calibration targets of the type shown in Figure 1B are advantageously processed by a method according to the invention insofar as they further minimize bias otherwise introduced by a conventional edge detection techniques. In this regard, it will be appreciated that such bias is reduced by the fact that "opposing" adjoining edges (i.e., edges that oppose one another across the reference point) define straight linear segments that change polarity across the reference point. That is, those segments are defined by regions that transition -- preferably, equally in magnitude -- from light-to-dark one one side of the reference point, and from dark-to-light on the other side. This is true for all "symmetric" calibration targets according to the invention, i.e., targets in which opposing edges define straight linear segments that are opposite polarity on either side of the reference point.

On the second and subsequent invocations of step 120, the method specifically applies the caliper tool at points along the lines found in the previous iteration. Preferably, these points are at every pixel in the image that lies along the line, though, those skilled in the art will appreciate that few points can be used. The calipers can be applied orthogonal to the lines, though, in the preferred embodiment, they are applied along the grid defined by the image pixels. The caliper range decreases with each subsequent invocation of step 120. The method continues applying the calipers, beginning at the estimated center point until one of the four following situations occurs: no edge is found by the caliper applied at the sample point; more than one edge is found by the caliper and the highest scoring edge is less than twice the score of the second highest scoring edge (this 2X comes from the CONFUSION_THRESHOLD); the distance between a computed edge point and the nominal line (computed from the previous invocation of step 120) is larger than a threshold (which threshold decreases with each subsequent invocation of step 120); or, the caliper extends outside of the image. As above, in step 120, the method then fits a line to the points found along each edge by the caliper tool, preferably, using a conventional least squares technique.

In the second and subsequent invocations of step 122, the method computes a refined location of the reference point as the intersection of the lines identified in step 120. As above, although conventional geometric calculations can be performed for this purpose, preferably, the reference point is computed using conventional least squares techniques.

Referring to Figure 5, there is also shown a machine methodology according to the invention for interpreting an image of a target 84 to determine its reference point and orientation. Particularly, in step 150, the method calls for generating an image of a target 84 and, particularly, of a target according to the invention having two or more regions, each region being defined by at least two linear edges that are directed toward a reference point, and having at least one of the regions having a different imageable characteristic from an adjacent region. Step 152 can be effected in the manner described in connection with step 100 of Figure 4, or equivalents thereof.

In step 152, the method analyzes the image to generate an estimate of an orientation of the target in the image. Step 152 can be effected in the manner described in connection with steps 102 - 108 of Figure 4, or equivalents thereof.

In step 154, the method analyzes the image to generate estimate of a location of the target's reference point. Step 154 can be effected in the manner described in connection with steps 110 - 118 of Figure 4, or equivalents thereof.

In step 156, the method analyzes the image to refine its estimates of the location of the reference point in the image and the orientation of the target in the image. Step 156 can be effected in the manner described in connection with steps 120 - 122 of Figure 4, or equivalents thereof.

Calibration target and methods for analysis according to the invention are advantageous over prior art targets and methods insofar as they are magnification invariant. By analyzing the adjoining edges of targets, methods according to the invention insure reliance on features (to wit, regions) that retain the same imageable appearance regardless of magnification. This is in contrast to prior art targets and methods, which rely on individual lines (or dots) to define calibrating features. As noted above, the imaging appearances of such lines and dots change with varying degrees of magnification. Even the prior art methods that analyze checkerboard targets rely on analysis of corners, which are not magnification invariant.

The magnification invariance of targets and methods according to the present invention is illustrated in Figures 6A - 6C. Referring to Figure 6A, there is shown an imaging setup wherein camera 200 images a target 202 (on object 204) from a height x. An image generated by camera 200 is displayed on monitor 206 of workstation 208. Referring to Figure 6B, there is shown an identical imaging setup, except insofar as a camera (of identical magnification) images a target (of identical size) from a greater height, x'. Likewise, Figure 6C shows an identical imaging setup, except insofar as a camera (again, of identical magnification) images a target (again, of identical size) from a still greater height, x". Comparing monitor images depicted in Figures 6A - 6C, it is seen

that the images of the targets, generated by the cameras and displayed on the monitors, are identical in appearance regardless of the relative heights (x, x', and x") of the camera. This magnfication invariance results from the fact that the target has the same appearance regardless of height (or equivalently, magnification), within specified limits thereof. Those skilled in the art will appreciate that those limits are greater than the analagous limits for prior art targets (e.g., cross hairs, parallel stripes, etc.).

Still another advantage of calibration targets and methods according to the invention is that they permits angular orientation to be determined throughout a full 360° range. With reference to Figure 1D, for example, the relative positions of the regions can be used to determine the overall orientation of the target, i.e., whether it is rotated 0°, 90°, 180°, or 270°. This information can be combined with a determination of relative orientation made by analysis of the adjoining edges as discussed above to determine the precise position of the target.

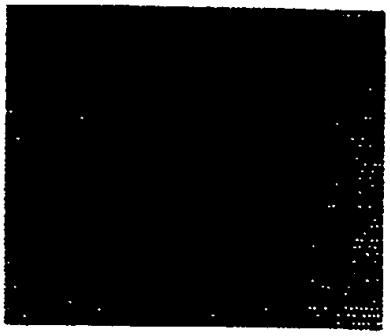
APPENDIX I

Patent Application for

MACHINE VISION CALIBRATION TARGETS AND METHODS OF DETERMINING THEIR LOCATION AND ORIENTATION

Vision Tool Description

THE FOLLOWING APPENDIX IS NOT BELIEVED TO BE NECESSARY FOR ENABLEMENT OR BEST MODE DISCLOSURE OF THE INVENTION DISCLOSED AND CLAIMED IN THE ACCOMPANYING APPLICATION.



Edge Detection Tool

This chapter describes the Edge Detection tool, which includes edge detection and peak detection.

Edge detection takes an input image and produces two output images: an image of the edge magnitude of each input pixel and an image of the edge angle of each input pixel. When combined with other Cognex software tools, the information produced by edge detection can be used to locate objects within an image. Edge pixel information can be used to detect the rotation of an object or defects such as scratches, cracks, or particles. You can also gauge how sharply an image is focused with edge pixel information.

Peak detection is useful in any application in which knowledge of the local maximum values in a two-dimensional space is useful. Peak detection takes an input image and produces an output image containing only those pixels in the input image with higher values than neighboring pixels. A typical input image to peak detection is an edge magnitude image; the output image contains only the highest magnitude edge pixels. The edge detection function can optionally use peak detection to postprocess the edge detection results so that only the strongest edges remain.

This chapter has nine sections as follows:

An Overview of Edge Detection describes the goals of edge detection, defines an edge pixel, and describes how the Edge Detection tool finds edge pixels. This section also explains the two properties of a pixel that are calculated by the Edge Detection tool, edge magnitude and angle, and ends with a sample application.

Using Edge Detection describes, in general terms, the interface to the Edge Detection tool. This section contains a discussion of the compression tables that affect the amount of memory used by the edge detector.

Edge Detection Enumerations and Data Structures describes the data structures and enumerations that the edge detection functions use. Types and data structures that support peak detection are discussed in *Peak Detection Enumeration Types and Data Structures* below.

4 Edge Detection Tool

Edge Detection Functions describes the functions that implement edge detection.

Maximizing the Performance of VC2 Vision Coprocessor lists the edge detection parameter settings that you should avoid if you want to maximize the performance of the VC2 vision coprocessor during edge detection.

Overview of Peak Detection defines peak pixel and gives examples of some applications of peak detection.

Using Peak Detection describes the interface to peak detection.

Peak Detection Enumeration Types and Data Structures describes the peak detection enumerations and data structures.

Peak Detection Functions describes the peak detection functions.

Some Useful Definitions

edge pixel	Pixel that has a different value from one or more of its eight neighboring pixels.
edge detector	Operator that finds edge pixels in an image.
compression tables	Two tables that are created during edge detection initialization: the edge compression table and the magnitude compression table. These tables are used by the Edge Detection tool to map data into a smaller range of values.
Sobel operators	Two 3x3 operators that the Edge Detection tool uses to locate edge pixels in an image.
horizontal edge component	Value returned by the horizontal Sobel operator when it is applied to a pixel in an image.
vertical edge component	Value returned by the vertical Sobel operator when it is applied to a pixel in an image.
edge magnitude	Value that increases as the difference in grey levels between neighboring pixels increases.
edge angle	Orientation of an edge, with respect to the x-axis of the image, at an edge pixel. The Edge Detection tool defines the angle of a pixel as arctan(v/h), where v is the vertical edge component and h is the horizontal edge component of the pixel.
peak pixel	Pixel with a value greater than or equal to some or all of its neighboring pixels' values.
peak detection	Finding some or all peak pixels in an image.
8-point neighborhood	Pixels that are horizontally, vertically, and diagonally adjacent to a pixel.
4-point neighborhood	Pixels that are horizontally and vertically adjacent to a pixel.
2-point neighborhood	Neighbors of a pixel along a single axis. Peak detection supports a horizontal and a vertical 2-point neighborhood operator.

4 Edge Detection Tool

symmetric peak	Peak whose pixel value is greater than or equal to the values of each of its neighbors.
asymmetric peak	Peak whose pixel value is greater than the values of its left and lower neighbors and greater than or equal to the values of its right and upper neighbors.
all-axes peak	Pixel that satisfies the peak definition (symmetric or asymmetric) along all axes in its neighborhood.
single-axis peak	Pixel that satisfies the peak definition (symmetric or asymmetric) along at least one axis in its neighborhood.
plateau	In peak detection, a region of contiguous pixels of the same value.

An Overview of Edge Detection

The Edge Detection tool finds edge pixels in an image. This overview of edge detection contains the following descriptions:

- Definition of the problem of edge detection
- Overview of the tasks that the Edge Detection tool performs
- Definition of edge pixel
- Description of the Sobel operator, which is used by the Edge Detection tool to locate edge pixels
- Definitions of the angle and magnitude of an edge pixel, and descriptions of the two images produced by the Edge Detection tool: the *magnitude image* and the *angle image*
- Description of edge detection preprocessing
- Description of edge detection postprocessing
- Description of a sample application that uses both the magnitude and the angle images to create an angle "signature" of a shape

Edge Detection

The Edge Detection tool locates edges and determines each edge's angle and magnitude.

An edge occurs wherever adjacent areas of an image have different grey values. Using this definition the triangle in Figure 66 has a single edge.

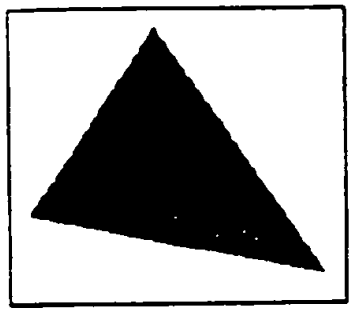


Figure 66. A figure with an edge

4 Edge Detection Tool

The definition of edge can be further refined as the directed border between grey areas of an image, where *direction* is defined as a vector normal to the edge. Using this definition, the triangle in Figure 67 has three directional edges, represented by the three vectors. The *angle* of the edge is the counterclockwise angle of the vector normal to the edge, with respect to the horizontal axis of the image. See the section *Angle Image* on page 181 for a discussion of edge angle.

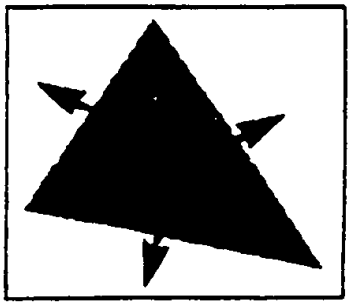


Figure 67. Vectors indicating the edge angles of a triangle

Along with an angle, an edge has a *magnitude*, which reflects the amount of the difference between grey levels on either side of the edge. As the difference in grey levels increases, the magnitude increases. See the section *The Magnitude Image* on page 178 for a discussion of edge magnitude.

The triangle in Figure 68 has the same three edges as the triangle in Figure 67; however, these edges are of lower magnitude. This lower magnitude is illustrated by the shortened length of the three direction vectors.

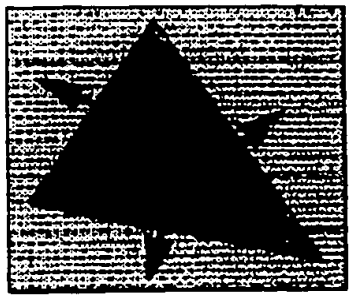


Figure 68. A triangle with low-magnitude edges

Edge Detection Tool 4

Edges are actually located pixel by pixel. Figure 69 contains a magnified view of a portion of an edge in an image. Each cell in the figure represents the grey level of a single pixel. Notice that, on this "microscopic" level, there are many edges between pixels in which the grey levels on either side of the edge differ by a few percentage points.

5	6	6	4	3	4	4	4	10	11	11
2	4	4	4	4	4	5	10	8	23	20
4	4	4	4	2	5	12	20	20	45	46
6	4	3	8	8	21	18	38	33	45	45
8	9	8	22	20	40	45	45	44	45	46
9	22	24	45	45	49	51	45	47	46	46
22	45	45	43	45	45	50	45	46	41	46
44	50	45	46	45	45	45	44	48	57	43
44	51	45	45	48	46	46	42	44	48	45

Figure 69. Grey levels in a magnified portion of an edge

The Edge Detection tool locates edges in an image by identifying each pixel that has a different value from one of its neighboring pixels. It calculates the angle and magnitude of each edge pixel. It also provides a means of classifying edges, so that low-magnitude edges are not reported.

4 Edge Detection Tool

Edge Pixels

A pixel is an *edge pixel* if it has a different value from at least one of its eight neighboring pixels and is not on the border of an image. In Figure 70, the four shaded pixels are edge pixels.

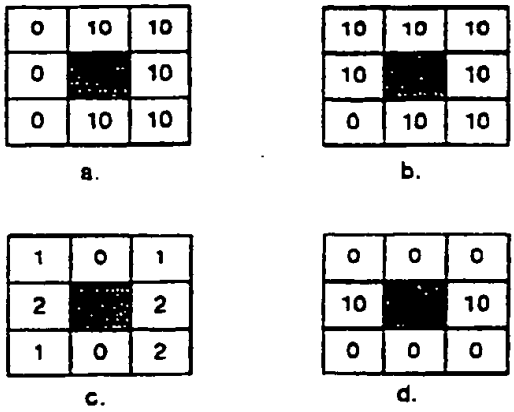


Figure 70. The shaded pixels are edge pixels

Figure 71 contains a grid representing grey levels in an image with the highest magnitude edge pixels shaded. Notice that border pixels (pixels along the edge of the image) are not edge pixels.

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	3
0	0	0	0	0	0	0	1	2	4	4
0	0	0	0	0	1	2	4	4	4	4
0	0	0	1	2	4	4	4	4	4	4
0	1	2	4	4	4	4	4	4	4	4
3	4	4	4	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4	4	4	4

Figure 71. Highest magnitude edge pixels in an image

The Edge Detection Operators

The Edge Detection tool finds edge pixels using the *Sobel* 3x3 neighborhood operators. During edge detection, the Sobel operators are applied to each pixel in an input image. This operation produces two values for each pixel. One value represents the *vertical edge component* for the pixel; the other value represents the *horizontal edge component* for the pixel. These two values are then used to compute the *edge magnitude* and *edge angle* of the pixel.

Sobel edge detection uses two 3x3 neighborhood operators to locate edge pixels in an image. The horizontal Sobel operator detects the horizontal (x) edge component for a pixel. The vertical Sobel operator detects the vertical (y) edge component for a pixel. Figure 72 shows the Sobel operators.

-1	0	1
-2	0	2
-1	0	1

Horizontal operator

1	2	1
0	0	0
-1	-2	-1

Vertical operator

Figure 72. Sobel operators

The Sobel operator works only on pixels with eight neighboring pixels. When the Sobel operator is applied to a pixel on the border of an image, the result is defined to be 0 because there are not enough neighboring pixels to calculate an edge value.

The horizontal and vertical Sobel operators are 3x3 linear operators. These operators compute a value for a pixel (x,y) in the following way:

- The grey level of the pixel (x,y) is multiplied by the value in the center of the 3x3 linear operator.
- The grey level of the neighboring pixel ($x-1,y-1$) is multiplied by the first value in the top row of the 3x3 linear operator.
- The grey level of the neighboring pixel ($x,y-1$) is multiplied by the second value in the top row of the 3x3 linear operator.
- This continues until the product of each pair of values is calculated.
- These products are then summed to produce the output. For the Sobel operator the value is interpreted as horizontal or vertical edge component of the pixel.

4 Edge Detection Tool

If the result of applying at least one of the Sobel operators to a pixel is nonzero, that pixel is an edge pixel.

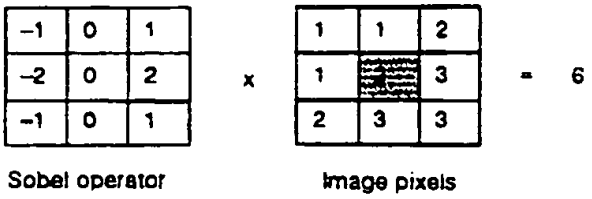


Figure 73. Applying the Sobel operator

In Figure 73, the horizontal Sobel operator is applied to the shaded pixel in the image. The resultant edge pixel value is 6, which is calculated as follows:

$$-1 \cdot 1 + 0 \cdot 1 + 1 \cdot 2 + -2 \cdot 1 + 0 \cdot 2 + 2 \cdot 3 + -1 \cdot 2 + 0 \cdot 3 + 1 \cdot 3 = 6$$

Note: The range of pixel values in Figure 73 and in most examples in this document is much smaller than the typical range of pixel values in an image. This is done to simplify the examples.

Figure 74 shows all the vertical and horizontal edge component values for a small image. The upper grid is the image; the values represent the grey levels of each pixel. The grid to the lower left of the image contains horizontal edge component values for the image. Each

cell in the grid contains the result of applying the Sobel horizontal operator to the corresponding input image pixel. The grid to the lower right contains the vertical edge components for the image. It is computed using the Sobel vertical operator.

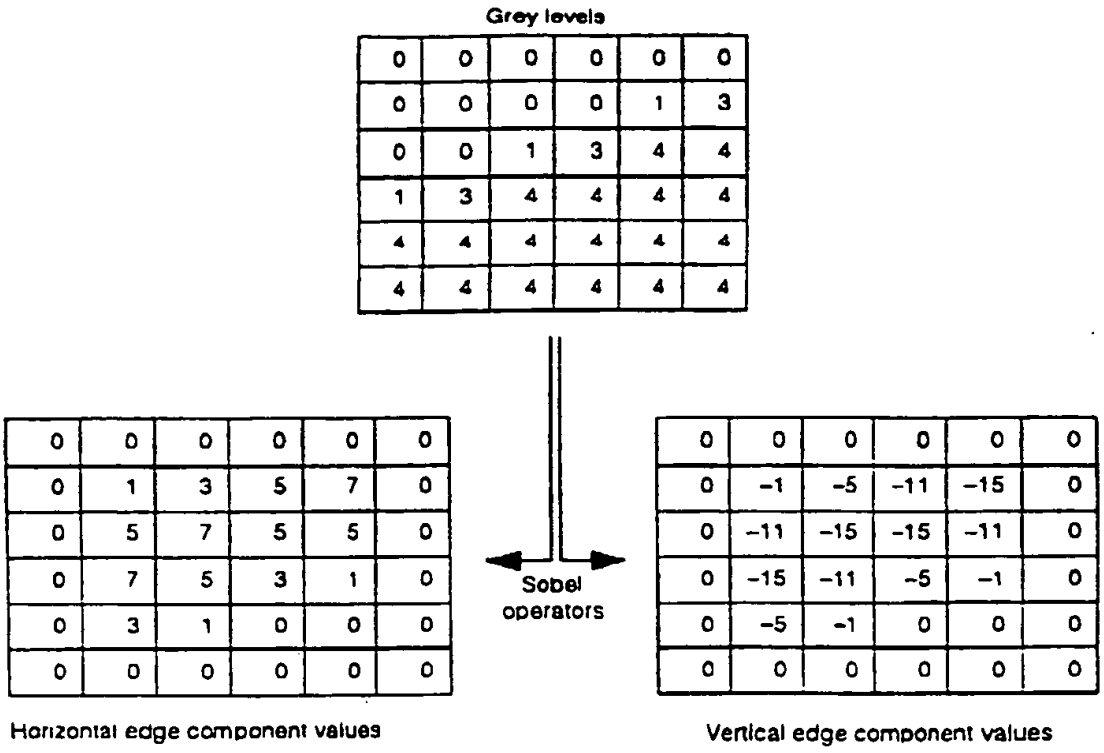


Figure 74. Horizontal and vertical edge component values computed with the Sobel operators

The horizontal and vertical edge component values computed with the Sobel operator are not returned by the edge detection functions. They are used to compute the edge magnitude and edge angle of each pixel in the image.

4 Edge Detection Tool

The Output Images

Edge magnitudes are stored in an output image that shows the edge magnitude of each corresponding pixel in the input image. Edge angles are stored in another output image that shows the edge angle of each pixel. You can choose which output images to create: magnitude, angle, or both.

These two images are described in this section.

The Magnitude Image

The edge magnitude for a given pixel is a function of the horizontal and vertical edge components of the pixel. If *x* is the horizontal edge component value and *y* is the vertical edge component value for some pixel, then the edge magnitude, *M*, for that pixel is defined as follows:

$$M = \sqrt{x^2 + y^2}$$

This formula suggests a geometric interpretation of the data: if *x* is the horizontal edge component and *y* is the vertical edge component, the magnitude is a vector from the origin of a two-dimensional coordinate system to the point (x,y). Figure 75 shows the geometric relationship between the edge magnitude and the two edge components.

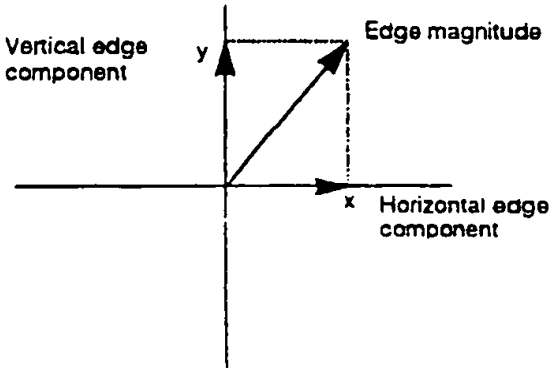


Figure 76. Geometric representation of edge magnitude

Note: For implementation-specific reasons, the Edge Detection tool uses the formula described above to compute edge magnitude and then scales the magnitude upward by approximately 16%.

You can choose the number of bits that can be used to express magnitude. Because magnitude values, computed with vertical and horizontal edge component values, can exceed this number of bits, magnitude values are *compressed* to an integer value that can be expressed in the number of bits you have specified. You can control how magnitude values are compressed: you can choose a method supplied by the Edge Detection tool, such as a logarithmic or linear map, or you can supply a magnitude compression map of your own. For descriptions of the maps supplied by the Edge Detection tool and for information on using your own map, see the section *Two Compression Tables* on page 192.

The size and depth of the magnitude image are the same as that of the input image. Since all border pixels have horizontal and vertical edge pixel values of 0, the border of the magnitude image contains all zeros. You can display an edge magnitude image: higher magnitude edges are brighter.

Optionally, edge detection can return the number of edge pixels in the input image and the sum of all edge magnitude values. By dividing the sum of all edge magnitude values by the size of the image, you can compute the average edge magnitude of an image. If you have two images of the same scene, you can determine which image is in sharper focus by comparing average edge magnitudes of the two images; the image with the higher average magnitude is in sharper focus.

4 Edge Detection Tool

Figure 76 contains an image, the image's horizontal and vertical edge components computed with Sobel operators, and the output image containing the edge magnitude values for each pixel. The edge pixels with the highest magnitude, representing the actual edge, are shaded.

0	0	0	0	0	0
0	0	0	0	1	3
0	0	1	3	4	4
1	3	4	4	4	4
4	4	4	4	4	4
4	4	4	4	4	4

Grey levels

0	0	0	0	0	0
0	1	3	5	7	0
0	5	7	5	5	0
0	7	5	3	1	0
0	3	1	0	0	0
0	0	0	0	0	0

Horizontal edge component values

0	0	0	0	0	0
0	−1	−5	−11	−15	0
0	−11	−15	−15	−11	0
0	−15	−11	−5	−1	0
0	−5	−1	0	0	0
0	0	0	0	0	0

Vertical edge component values

0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.6	6.7	14.1	[illegible]	0.0
0.0	14.1	[illegible]	[illegible]	14.1	0.0
0.0	[illegible]	14.1	6.7	1.6	0.0
0.0	6.7	1.6	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

Sobel edge magnitude values

Figure 76. Edge magnitude values computed with the Sobel operators

Angle Image

The edge detection functions produce an image containing the angle of each edge pixel. If *x* is the horizontal edge component value and *y* is the vertical edge component value, the edge angle is the counterclockwise angle between the horizontal axis and the magnitude vector.

Figure 77 shows the geometric relationship between the edge angle, the edge magnitude, and the two edge components. In this figure, *M* is the magnitude vector and θ is the edge angle.

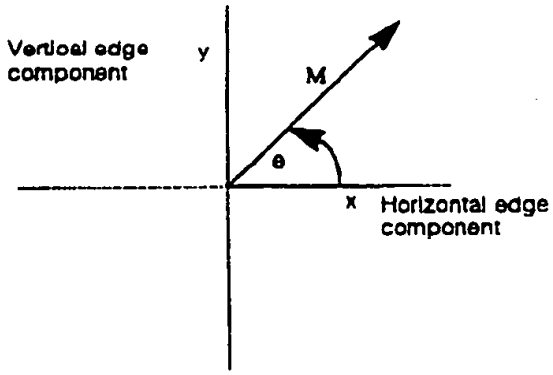


Figure 77. Geometric interpretation of edge angle

Edge angles are stored in the angle image in binary form. They are represented as fixed-point numbers with the decimal point to the left of the most significant bit and are interpreted as fractions of 360°. You can choose the maximum number of bits available to express an angle; this value must be less than or equal to 8. If *n* is the maximum number of bits used to express angle, and *x* is the binary representation of an angle, you can convert *x* to degree notation by using the following formula:

$$\text{angle}(x) = \frac{360x}{2^n}$$

Note that with a 6-bit representation, angles can be expressed to the nearest 5.625°; with a 7-bit representation, an angle can be expressed to the nearest 2.8°; with an 8-bit representation, an angle can be expressed to the nearest 1.4°.

4 Edge Detection Tool

The edge angle is computed using the arc tangent function. If *x* is the horizontal edge component and *y* is the vertical edge component for a pixel, the edge angle for that pixel is calculated as shown in Table 3.

If	If	Edge Angle
x > 0	y > 0	arctan(y/x)
x > 0	y < 0	360 – arctan(|y/x|)
x > 0	y = 0	0
x < 0	y > 0	180 – arctan(|y/x|)
x < 0	y < 0	180 + arctan(y/x)
x < 0	y = 0	180
x = 0	y > 0	90
x = 0	y < 0	270
x = 0	y = 0	0

Table 3. Formulae for computing edge angles

The computation is designed so that an edge angle θ is in the following range:

$$0° \leq \theta < 360°$$

Figure 78 contains six rows. Each row contains (from left to right) an image with the central pixel shaded, the vertical and horizontal edge component value of the shaded pixel, the image with a vector superimposed showing the edge angle of the central pixel, and the formula used to compute the edge angle. Notice that the equations defining the edge angle value are designed so that the vector always points to the brighter side of the edge.

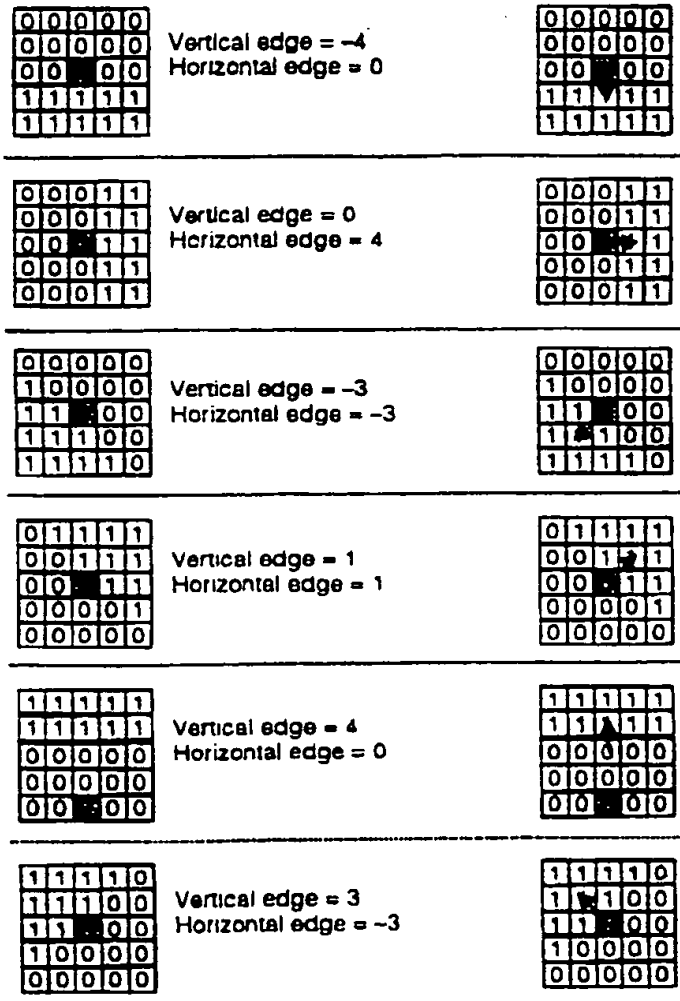


Figure 78. Computing the angle of various edges

183

4 Edge Detection Tool

By setting a flag in the edge detection parameters, you can specify that you want angles computed in the range 0° to 180° instead of the default range, 0° to 360°. When you select the smaller range, an edge angle *n* that is greater than 180° maps to edge angle (*n* — 180°). For instance, 300° maps to 120°, and 270° maps to 90°.

Use the reduced angle range when you want the same edge angle results regardless of the polarity of the image. As shown in Figure 79, when the range of angles is restricted to the range 0° to 180°, the same edge angles are computed for a black object on a white background as for a white object on a black background. When the range of angles is 0° to 360° (also Figure 79) this is not the case: reversing the polarity of the image reverses the edge pixel angle.

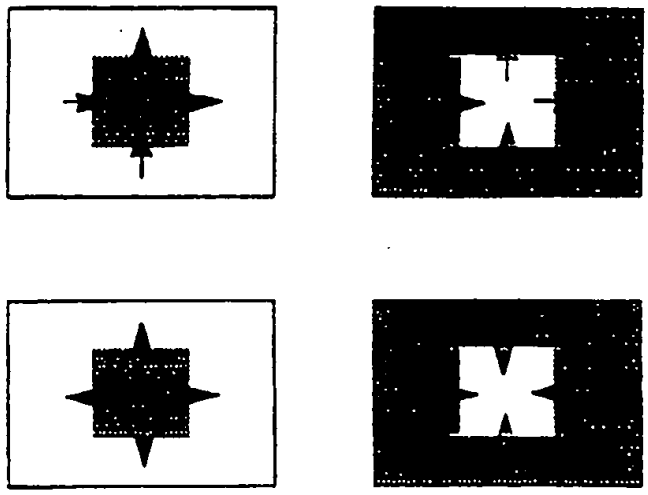


Figure 79. Effect of reversed polarity on edge angles for the two ranges

The binary representation of an angle in the range 0° to 180° is similar to the representation of an angle in the range 0° to 360°. If *n* is the maximum number of bits used to express angle and *x* is the binary representation of an angle in the range 0° to 180°, you can convert *x* to degree notation by applying the following formula:

$$\text{angle}(x) = \frac{180x}{2^n}$$

Edge Detection Preprocessing

If you supply a pixel map, the edge detection function uses it to preprocess your input image before it calculates the horizontal and vertical edge components. For a description of pixel mapping, see *Chapter 3, Pixel Mapping*.

Edge Detection Postprocessing

Although edge angles are defined for all pixels in an image, they are significant only at the pixels with the greatest edge magnitudes. There are many edges between pixels where the grey levels on either side of the edge differ by only a few percent. These insignificant edge pixels, due to frame grabber noise and random fluctuations in pixel values, can clutter your edge magnitude image and make it difficult to evaluate the data in that image. Also edge magnitude values usually increase gradually in a "smear" as they approach an edge and then decrease as the edge is crossed. You might want to sharpen the magnitude image by filtering out some edge pixels that are near, but not at, an edge.

The Edge Detection tool supplies three postprocessing methods. Two of these methods, *setup* and *run-time* thresholding, set the lowest edge magnitude pixels to 0. The third method, *peak detection*, takes an edge magnitude image and processes it so that, for any small pixel neighborhood, only the highest edge magnitude pixels remain and the other pixels are set to 0. See the section *Peak Detection on an Edge Magnitude Image* on page 229 for a complete discussion of this method.

After postprocessing, edge angles corresponding to zero edge magnitude pixels are also set to 0 unless you have specified run-time thresholding but *not* peak detection. In this case, edge angles corresponding to zero edge magnitude pixels can be nonzero, but should be considered *undefined*.

Setup Thresholding

You can supply a setup time rejection percentage. This is a value *n* that represents a percentage of the highest possible edge magnitude value. With this method, any magnitude values in the first *n* percent of all possible values are forced to 0. For example, if you provide 6 bits to express magnitude values and specify that the lowest 5 percent of magnitude values are forced to 0, then any magnitude below 3 is forced to 0 since 5 percent of 63 (truncated) is 3.

Run-Time Thresholding

You can supply a run-time rejection percentage. This is a value *n* that represents a percentage of the highest *actual* edge magnitude value in a magnitude image. With this method, magnitude values in the first *n* percent of the magnitude values in a specific image

are forced to 0. For example, if the highest magnitude value in a particular image is 51, and if you specify that the lowest 5 percent of the run-time magnitude values are forced to 0, then any magnitude below 2 is forced to 0, since 5 percent of 51 (truncated) is 2.

Using a setup rejection percentage is faster than using a run-time rejection percentage, since the values that are forced to 0 are computed once, before any magnitude images are created. However, the run-time method is more flexible and robust. If you are producing magnitude images of the same scene but under varying lighting conditions, you should use a run-time rejection percentage since the proportion of magnitude values forced to 0 remains stable as the range of magnitude values changes (due to varying illumination).

Postprocessing with Peak Detection on a Magnitude Image

Peak detection is a method of filtering an image so that only *peak pixels* remain and other pixels are set to 0. Peak pixels have a value greater than or equal to some or all of their neighboring pixels' values.

You can set a flag in the edge detection parameters structure that instructs the edge detection function to run peak detection on the magnitude image during postprocessing. All nonpeak pixels are filtered out of the magnitude image and the edge angle image before they are returned. When you specify this, you must supply peak detection parameters in the edge detection parameter structure.

The edge detection function does peak detection postprocessing after it has done setup and run-time thresholding. There will be a 2-pixel-wide border of zeros in a peak-detected magnitude image.

The section *Peak Detection on an Edge Magnitude Image* on page 229 discusses, in detail, the benefits of using peak detection on an edge magnitude image. This section also describes the proper way to parameterize peak detection so that it filters edge magnitude pixels correctly. However, you do not need a detailed knowledge of peak detection to use it with edge detection. Suggested peak detection parameters are supplied in the section *Data Structure cip_edge_params* on page 199.

Figure 80 contains an image, its edge magnitude image, and the image that results from peak detection. The edge magnitude image contains a smeared edge, which is sharpened by peak detection.

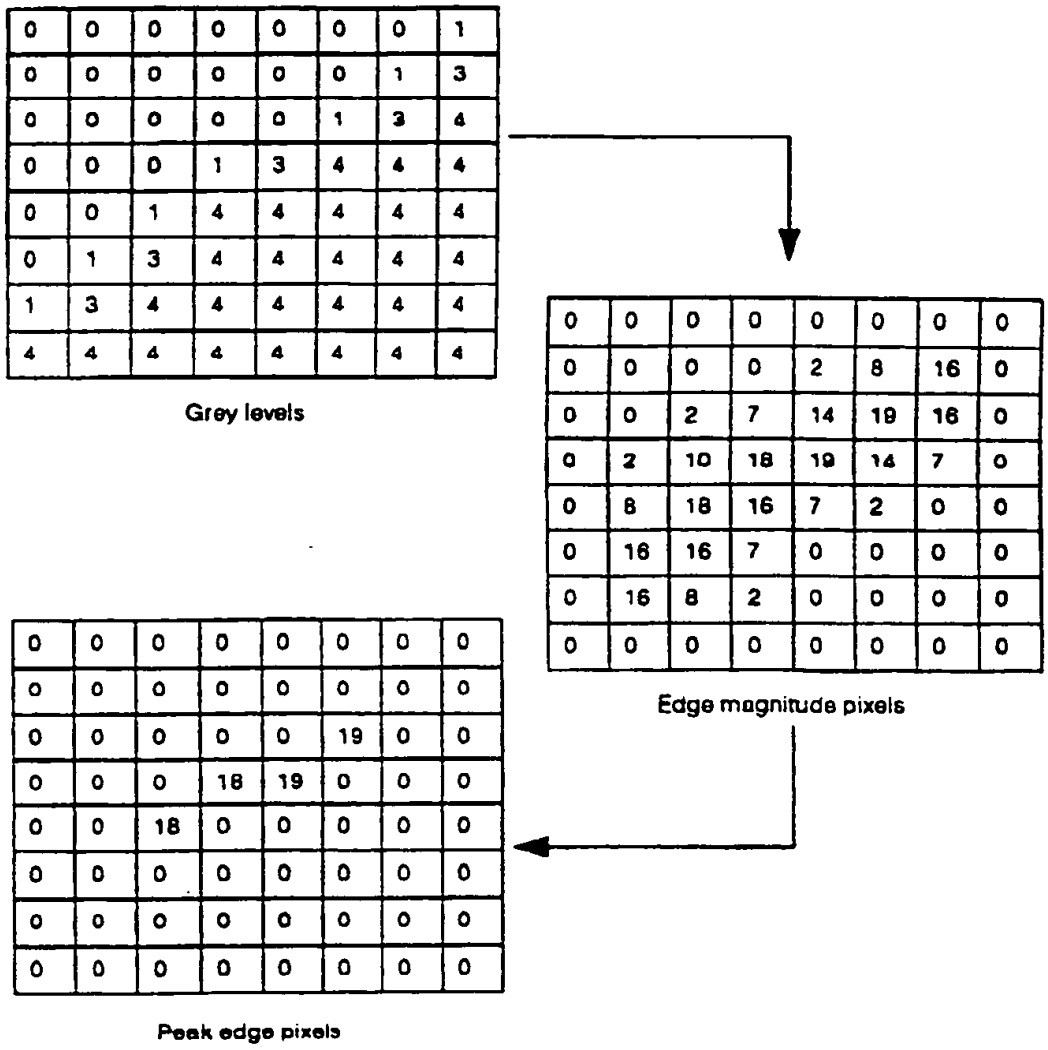


Figure 80. Peak detecting an edge magnitude image to determine actual edge

4 Edge Detection Tool

Sample Application: Angle Histogram

You can use the magnitude and angle images of an object to create an *angle histogram* of that object. This is a table that assigns a value *n* to each edge angle θ if there are *n* edge pixels (above a given magnitude threshold) whose edge angle is θ.

The angle histogram supplies a useful "signature" of an object's shape. You can use this signature to gauge similarity among objects, to detect rotations of an object, and to detect flaws in an object.

Once you have created edge magnitude and edge angle images for an object, you can create and display an angle histogram as follows:

```
void angle_histogram(res,threshold)
cip_edge_results *res;      /* results structure pointer */
int threshold;              /* significant edge threshold */
{
int i,j,mag;
int hist[256];

cu_clear(hist,sizeof(hist));
for (i=0; i<res->mag_img_p->width; i++)
    for (j=0; j<res->mag_img_p->height; j++){
        mag=res->mag_img_p->get(res->mag_img_p,i,j);
            if (mag > threshold)
                hist[res->ang_img_p->get(res->ang_img_ p,i,j)]++;
}
cgr_histogram(caq_image,hist,65);
caq_display();
return;
}
```

Figure 81 contains four shapes; each shape is paired with its angle histogram. Note that the square and the plus sign have identical angle histograms. The angle histogram of the circle is flat because there is no dominant angle along the edge of a circle.

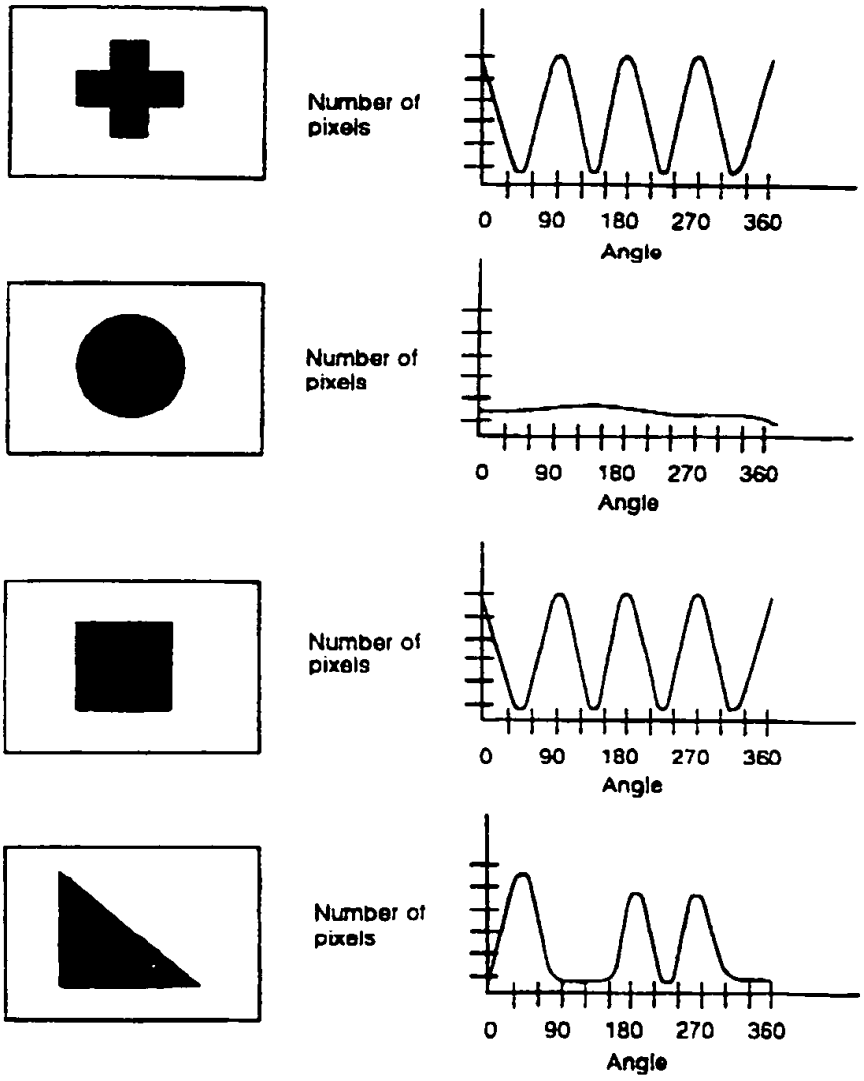


Figure 81. Assorted shapes with their angle histograms

4 Edge Detection Tool

Figure 82 demonstrates the effect of rotation on the angle histograms of a plus sign. The histogram shifts as the plus sign rotates.

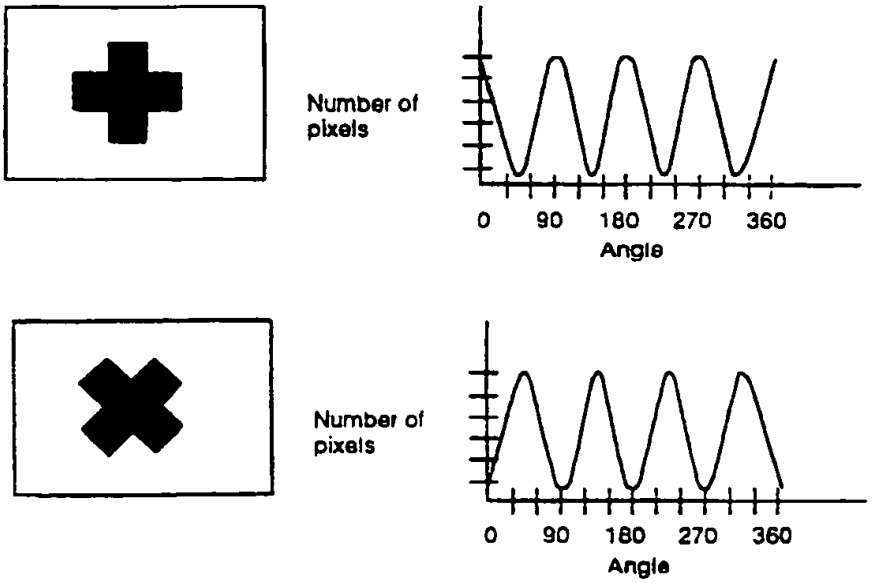


Figure 82. Effect of rotation on angle histogram

The Caliper Tool

This section defines the Caliper Tool and describes in general terms how it works. The concepts introduced in this section are described in detail in later sections.

The Purpose of the Caliper Tool

The Caliper Tool is a tool for locating edges and edge pairs in an image. The edge of an object in an image is a change in grey value from dark to light or light to dark. This change may span several pixels. The Caliper Tool provides methods for ignoring edges that are caused by noise or that are not of interest for a particular application.

The Caliper Tool is modeled on the mechanical engineer's caliper, a precision device for measuring distances. You specify the separation between the caliper "jaws," and the Caliper Tool searches an area you specify for edge pairs separated by that distance. You can also search for individual edges when you know their approximate location in an image. This is similar to using a caliper to measure depth.

A typical use for the Caliper Tool is part inspection on an assembly line. For example, in integrated circuit lead inspection, a part is moved in front of a camera to an approximately known location (the *expected* location). You use the Caliper Tool to determine the exact location of the left edge of the part by searching for a single light to dark edge at the expected location (see Figure 103). The edge that is closest to the expected location is the left edge of the part.

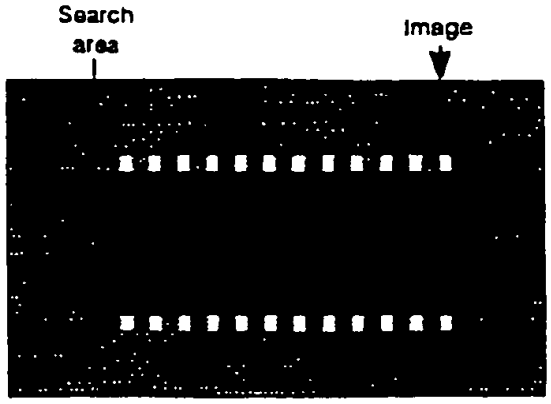


Figure 103. The Caliper Tool is used to locate the left edge of the part.

4 Caliper Tool

Once you find the left edge of the part, you can predict where the leads should lie within the image to specify a window in which to look for leads (see Figure 104). You can then use the Caliper Tool to search for edge pairs consisting of a dark to light transition followed by a light to dark transition, and separated by the expected lead width. With information calculated by the Caliper Tool, you can compare measured lead width and location to expected lead width and location to make an accept/reject decision about the part.

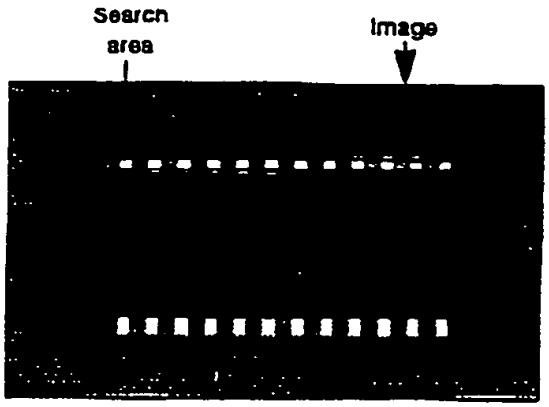


Figure 104. The Caliper Tool can be used to measure lead width and lead spacing on an integrated circuit.

How the Caliper Tool Works

The Caliper Tool uses *projection* to map a two-dimensional window of an image (the *caliper window*) into a one-dimensional image. Projection collapses an image by summing the pixels in the direction of the projection, which tends to amplify edges in that direction. Figure 105 shows an image, a caliper window, the one-directional image that results from projection, and a graph of the pixel grey values in the one-dimensional image.

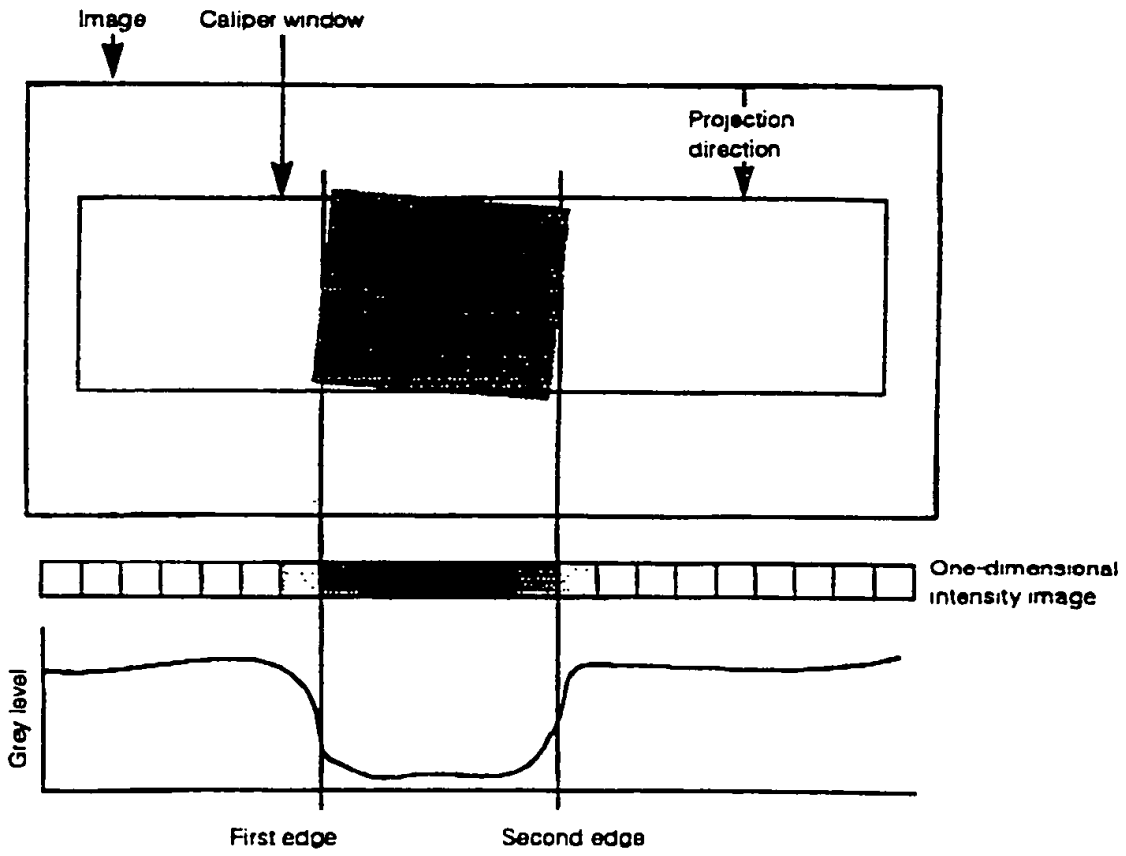


Figure 105. Overview of the Caliper Tool

You may specify that the Caliper Tool perform *pixel mapping* before projection. Pixel mapping can be used to filter an image, attenuate or amplify a range of grey values, and otherwise map pixel values. See *Chapter 3, Pixel Mapping*, in the *Image Processing* manual for a complete description of pixel mapping.

After projection, an *edge filter* is applied to the one-dimensional image to further enhance edge information and to smooth the image by eliminating minor grey value changes between neighboring pixels that are most likely caused by noise. The edge filter produces

4 Caliper Tool

a first derivative image (the most rapid changes in grey value in the projected image result in the highest peaks in the filtered image). Figure 106 shows the one-dimensional image from Figure 105, an image generated by the edge filter, and a graph of that image.

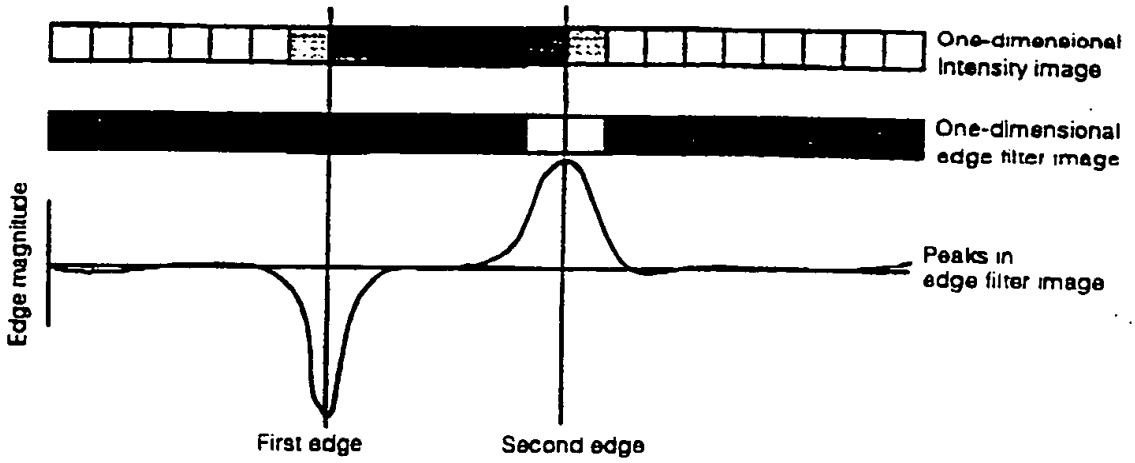


Figure 106. An edge filter is applied to the one-dimensional image that results from projection

After the filter operation, *edge detection* is performed on the filtered image. Edge detection is a method of evaluating peaks in the filtered image and ignoring edges that are not of interest, such as edges representing noise in the image. Edge detection applies *geometric constraints* that you specify to each edge or edge pair found in the image. Applying geometric constraints provides a way to limit the number of edges to evaluate by assigning a score to each edge or edge pair based on such factors as the expected location of the edge or edge pair, the distance between edges in an edge pair, and the minimum grey level difference between pixels on each side of the edge.

The Caliper Tool returns as many results as you have specified in the **cclp_params** data structure and computes the geometric average of the score of each geometric constraint that you specify. For edges or edge pairs whose score equals or exceeds the accept threshold, the Caliper Tool returns information such as the score and location of the edge or edge pair and the measured distance between edges in an edge pair.

The controlling variables of an edge or edge pair search define a *caliper*, and include such information as the dimensions of the search window and the edge detection scoring parameters. The type of the data structure that defines a caliper is **cclp_caliper**.

The Caliper Window and Projection

This section describes how you specify the *caliper window*, which is the subset of an image in which to locate edges or edge pairs. It then describes *projection*, which is the mapping of the caliper window into a one-dimensional image. Finally it describes the *edge filter*, which enhances edges in the one-dimensional image.

Caliper Window

The *caliper window* is the portion of an image in which the Caliper Tool searches for edges. It is defined by the *search length* and *projection length*, which are the width and height respectively, of the window in an image that will be projected into a one-dimensional image. The caliper window is illustrated in Figure 107.

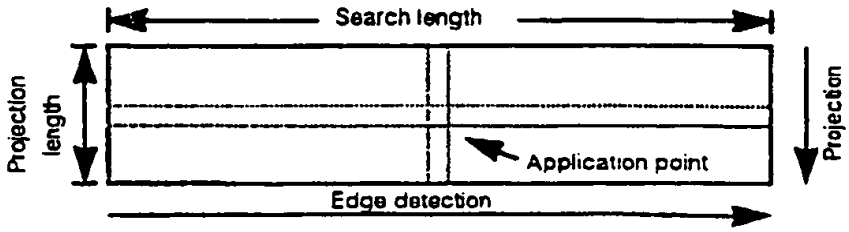


Figure 107. The Caliper Tool window

The pixel in the center of the caliper window is called the *application point*. This is the point in the run-time image at which you want to locate edges.

Caliper Window Rotation

For many applications edge information is needed at more than one angle in an image. For example, to inspect a rectangular part, you may want to measure the dimensions of the part by projecting the image first at 0° to measure length, and then at 90° to measure width. The caliper window can be oriented at any angle to locate edges in an image. Angles increase in the clockwise direction from the horizontal axis.

Figure 108 shows how changing the caliper window angle affects the projection and edge search directions at 90° rotations. The Caliper Tool runs faster for 90° rotations than it does for arbitrary rotations.

4 Caliper Tool

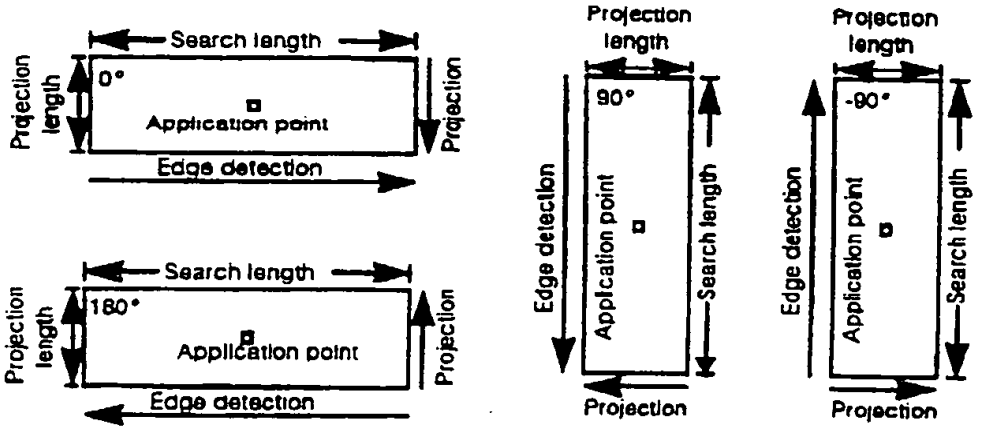


Figure 108. The caliper window at 90° rotations

Skewed Window Projection

Figure 109 shows a -15° caliper window with window rotation *disabled*. In this case, the Caliper Tool creates a "skewed" window and projects it along the angle of skew into a one-dimensional image.

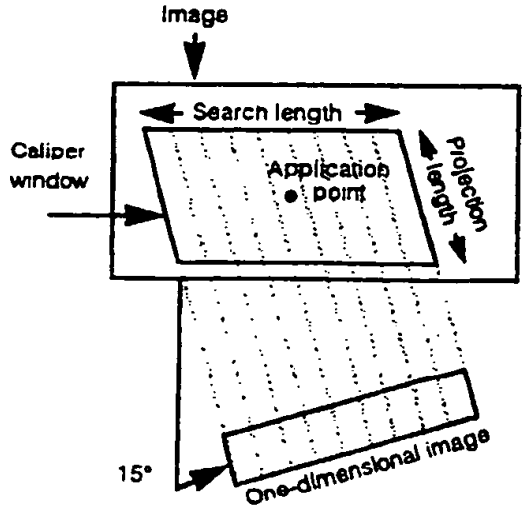


Figure 109. Skewed window projection

When window rotation is disabled, you specify how the skewed window will be projected: with or without *interpolation*. Interpolation increases accuracy at the expense of execution speed.

When you use a skewed window without interpolation for projection, the pixels in the source image are summed along the angle of skew as shown in Figure 110. Each pixel in the two-dimensional image contributes to one pixel in the one-dimensional image. In Figure 110, those pixels containing the number 1 contribute to the destination pixel containing the number 1, and so on.

4 Caliper Tool

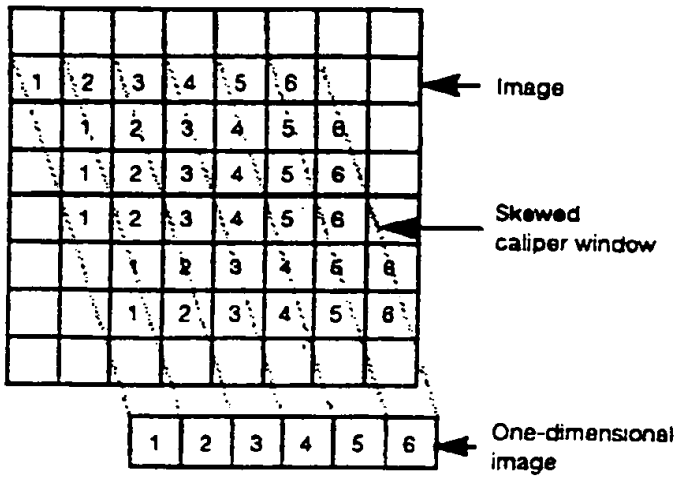


Figure 110. Skewed window projection without interpolation

When you use a skewed window with interpolation for projection, the skewed window is transformed into a rectangular window before projection. The transformation is performed by **clp_transform()**, which uses several neighboring pixels from the two-dimensional image to calculate the pixel value for the one-dimensional image. See *Chapter 2, Basic Image Processing*, in the *Image Processing* manual for a complete description of the function **clp_transform()**.

Rotated Window Projection

Figure 111 shows a -15° caliper window with window rotation *enabled*. In this case the Caliper Tool rotates the two-dimensional image and projects it into a one-dimensional image.

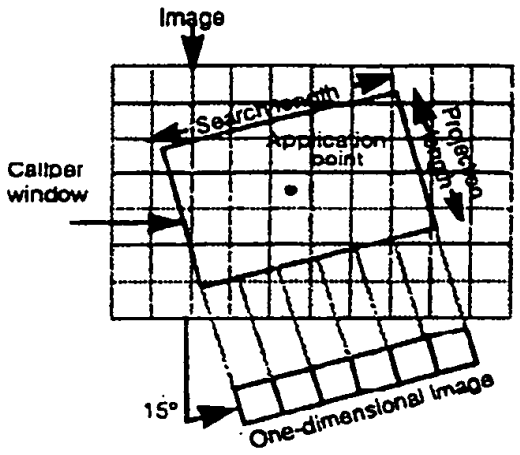


Figure 111. Rotated window projection

Choosing the Projection Type

The choice of *skewed* or *rotated* projection depends on the angle of the edge or edge pair of interest in the image and the remaining content of the image. If the rotation angle is close to 90° or if a skewed image will contain enough of the edge or edge pair of interest to generate strong edges, you can use *skewed projection* to speed program execution. Figure 112 shows an image, a caliper window and the one-directional image that results from projection. The skewed window contains enough edge information, so skewed projection may be used.

4 Caliper Tool

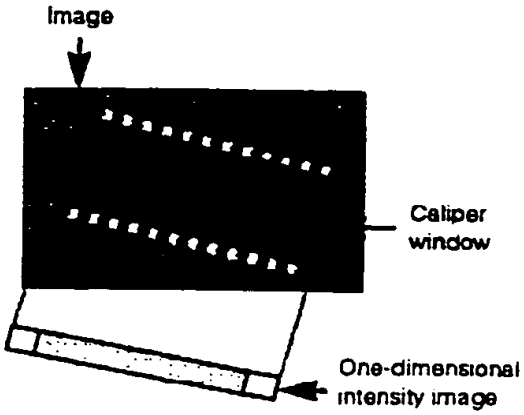


Figure 112. Skewed projection may be used

Figure 113a shows a caliper window where the edges of interest are at 15° and window rotation is disabled. A window large enough to enclose these edges would also include much of the dark area in the image, which in some cases would result in a one-dimensional image where the edges are obscured.

Figure 113b shows a caliper window at -15° in the same image with window rotation enabled. Only the edges of interest are included in this window.

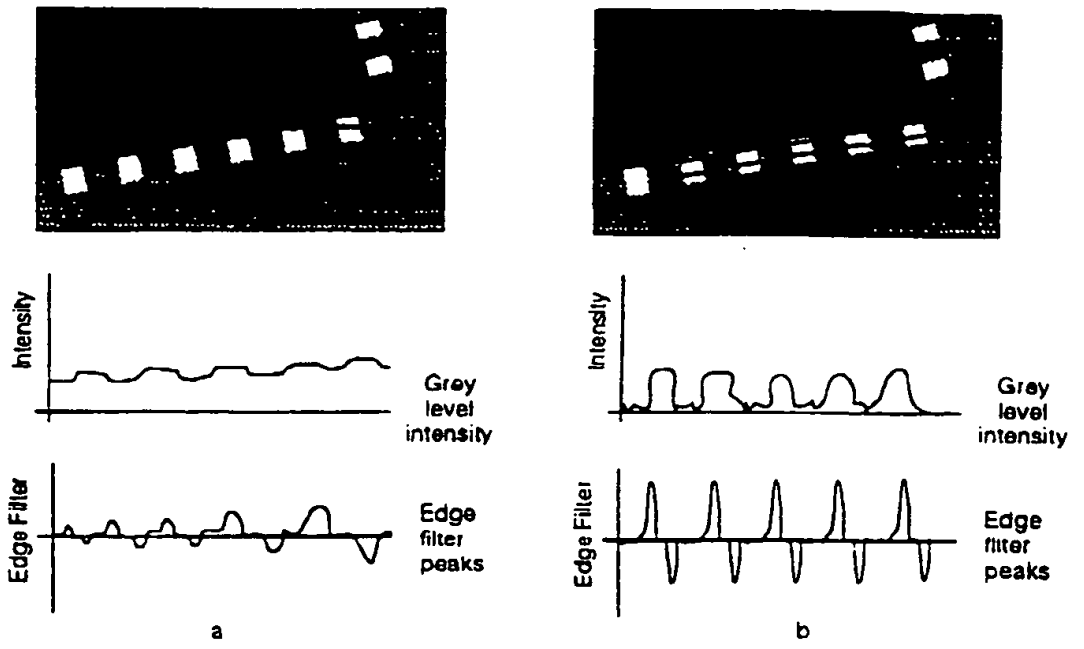


Figure 113. Skewed projection (a) and rotated projection (b)

Caliper Tool Optimization

When setting up the variables for a caliper, you specify the following types of optimization:

- *Space optimization:* The Caliper Tool is optimized for minimum memory use at the expense of execution speed.
- *Speed optimization:* The Caliper Tool will use a number of internal techniques to improve execution speed at the expense of increased memory use.
- *Default optimization:* The Caliper Tool will weigh space and speed equally, using slightly more space than if you specify space optimization and slightly slower execution speed than if you specify speed optimization.

4 Caliper Tool

The Edge Filter

After projection, an *edge filter* is run on the resulting one-dimensional image. The edge filter accentuates edges in the image and produces a filtered image. The peaks in this image indicate strong edges.

The two parameters of the edge filter are its *size* and *leniency*. Edge filter size is the number of pixels to each side of the edge to consider in the evaluation. Edge filter leniency is the number of pixels at an edge to ignore between light and dark sides of the edge (leniency is described in further detail below). Figure 114 illustrates an edge filter. The edge filter is positioned over the leftmost pixel in the one-dimensional image where it will entirely fit in that image. Pixel values to the left of the leniency region are summed and subtracted from the sum of the pixel values to the right of the leniency region: in this example, (0+5)-(0-0) yields 5 for the edge filter image.

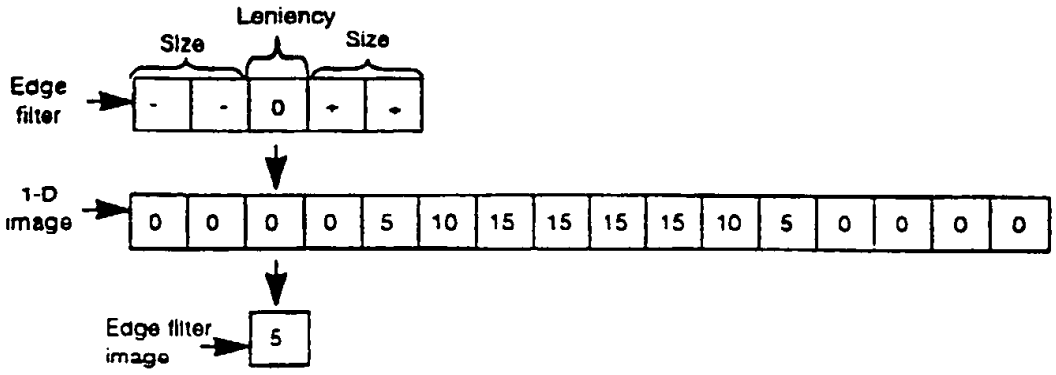


Figure 114. The edge filter is positioned in the one-dimensional image.

As shown in Figure 115, the edge filter is then applied at each successive pixel in the one-dimensional image until the edge filter no longer fits entirely in the image. At each pixel position, the pixel values to the left of the leniency region are summed and subtracted from

the sum of the pixel values to the right of the leniency region and the result is written to the edge filter image. The resulting image will be 2 * size + leniency - 1 smaller than the one-dimensional image.

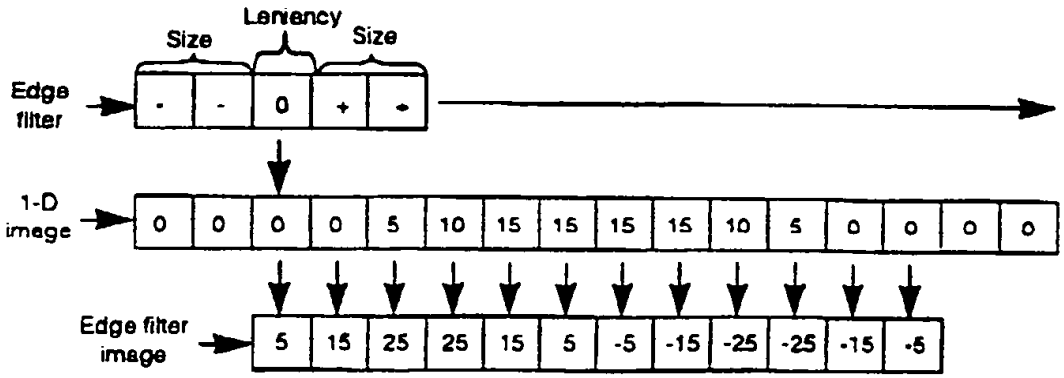


Figure 115. The edge filter is applied at each successive pixel until the edge filter no longer fits entirely in the image.

Figure 116 shows a graph of the one-dimensional image and the edge filter image from Figure 115. Peak position is calculated to subpixel accuracy.

4 Caliper Tool

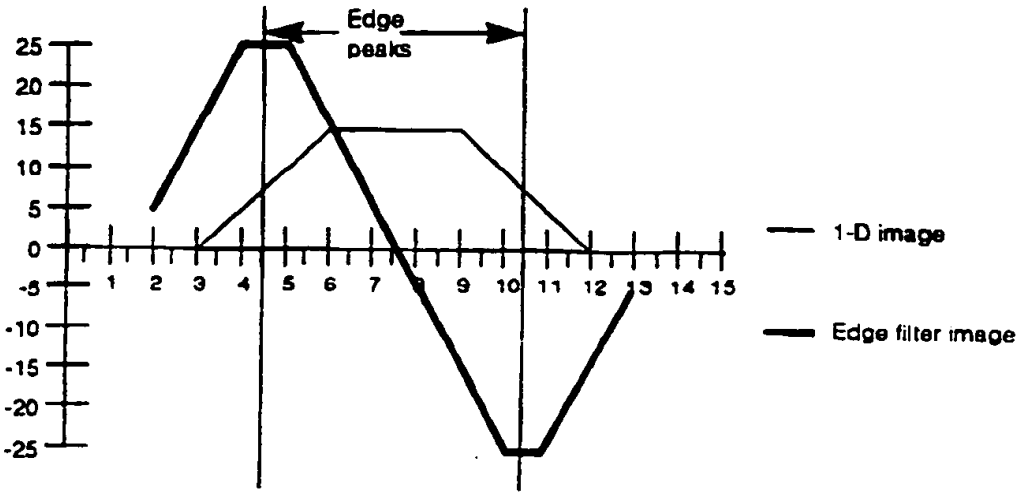


Figure 116. Edge filter image graph vs. one-dimensional image graph

The graph in Figure 116 represents an edge filter applied to an ideal image; few applications would result in as ideal a graph. All images contain some degree of noise, and a typical edge in an image can be several pixels wide. The next two sections provide guidelines for optimizing the edge filter by choosing appropriate values for size and leniency.

Choosing Edge Filter Size

Edge filter size specifies the number of pixels on either side of an edge to consider in an edge evaluation. Size should usually be greater than 1 because noise in an image causes grey level differences between neighboring pixels. Figure 117 shows a graph of a one-dimensional image of a light band on a dark background, and a graph of an edge filter image where a size of 1 was used. Because there is noise in the image, many more peaks exist than are of interest.

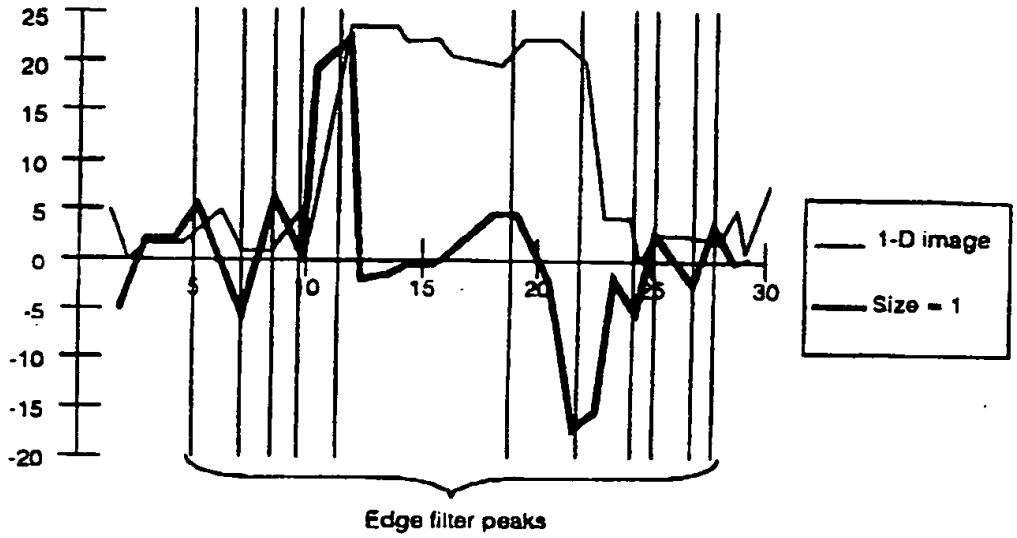


Figure 117. An edge filter size of 1 is typically too small.

Increasing size tends to smooth the edge filter image because summing several pixels on either side of an edge tends to reduce the effects of noise. Figure 118 shows an image similar to the one in Figure 117 where a size of four was used. The only two peaks in the image are the peaks of interest. Size is typically between 2 and 5 inclusive.

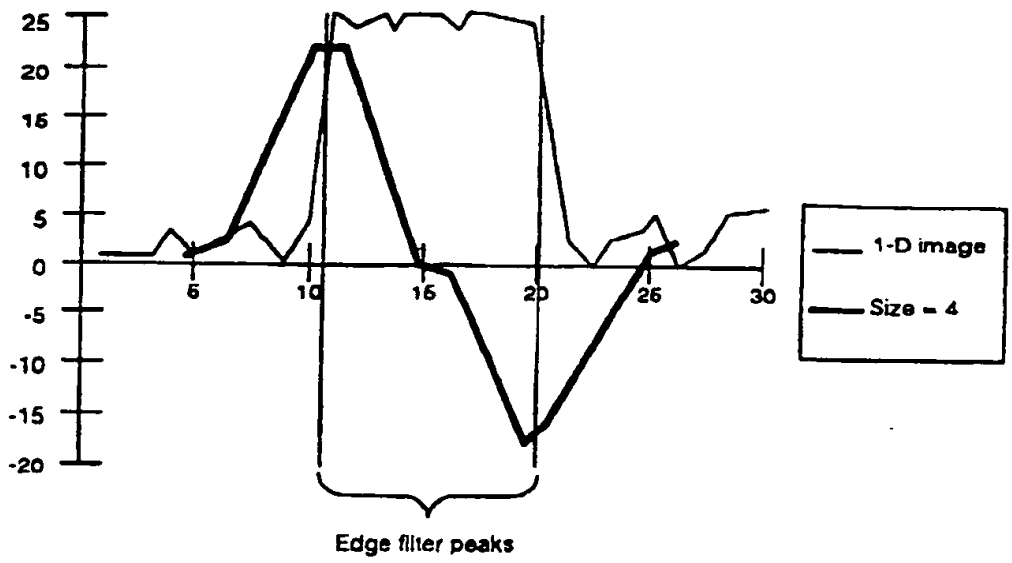


Figure 118. Using four for edge filter size reduces the effects of noise on the edge filter.

Choosing Edge Filter Leniency

Edge filter leniency specifies a region of pixels to ignore at an edge in an image. Due to slight changes in edge or edge pair position or orientation, a pixel on the edge of a feature may not always have the same grey value. By setting a *leniency* zone between the light and dark areas of a feature, the effects of slight changes in feature location on filter results are minimized.

Figure 119 shows a graph of a one-dimensional image with two edges that span two pixels each, and graphs of the output of two edge filters. The first edge filter has a leniency of 0, the second edge filter has a leniency of 2. The edge filter with a leniency of 2 produces higher peaks. Two is a typical value for leniency.

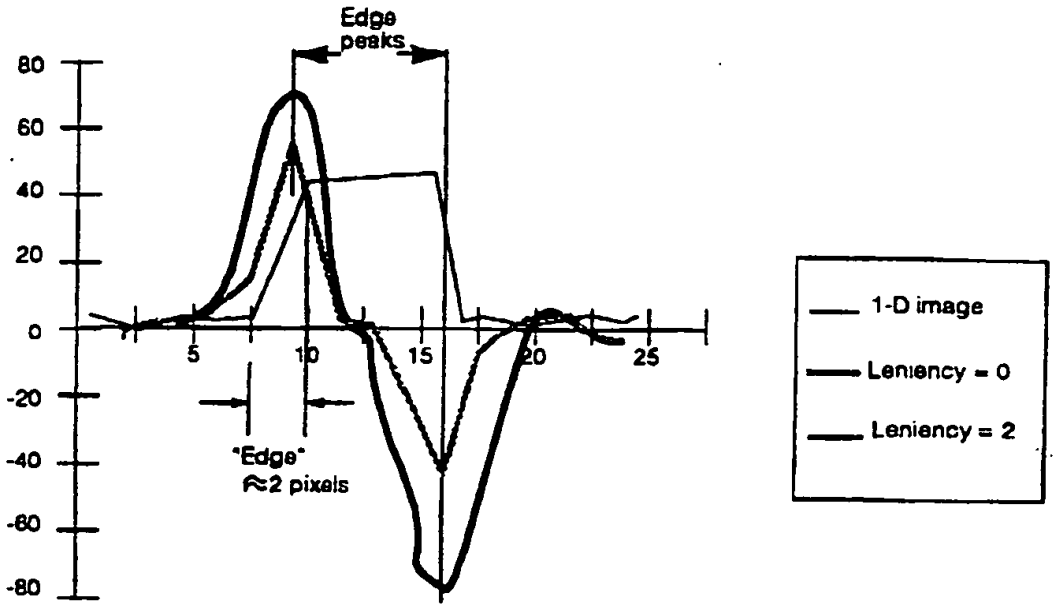


Figure 119. The effect of leniency on an edge filter image

Point Registration Tool

This chapter describes the Cognex Point Registration tool. It contains seven sections.

The first two sections, this one and *Some Useful Definitions*, provide an overview of the chapter and define some terms that you will encounter as you read.

Point Registration Tool Overview provides information about the capabilities and intended use of the Point Registration tool.

How the Point Registration Tool Works provides a general description of the operation of the Point Registration tool.

Comparing the Point Registration Tool with Search compares the Point Registration tool with the Cognex Search tool.

Using the Point Registration Tool describes some of the techniques that you will use to implement an application using the Point Registration tool.

Point Registration Tool Data Structures and Functions provides a detailed description of the data structures and functions that you will use to implement your application.

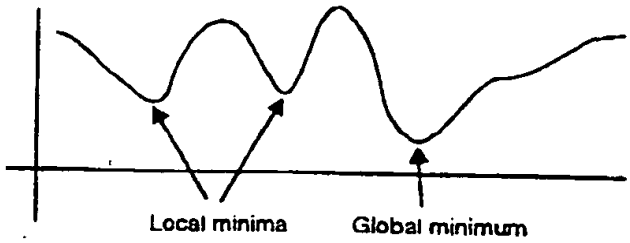
6 Point Registration Tool

Some Useful Definitions

point registration A search technique designed to determine the exact point at which two images of the same scene are precisely aligned.

global minimum The lowest value in a function or signal.

local minima A low value in a function or signal.



subpixel accuracy Positions within an image may be specified in terms of whole pixel positions, in which case the position refers to the upper-left corner of the pixel, or in terms of fractional pixel positions, in which case the position may lie anywhere within a pixel. Positions specified in terms of fractional pixel positions are referred to as having subpixel accuracy.

Point Registration Tool Overview

The Cognex Point Registration tool performs fast, accurate point registration using two images that you supply.

Point Registration

Point registration is the process of determining the precise offset between two images of the same scene.

You use the Point Registration tool to perform point registration by providing a model image and a target image, along with a starting point within the target image. The model image should be a reference image of the feature you are attempting to align. The target image should be larger than the model image, and it should contain an instance of the model image within it.

The Point Registration tool will determine the precise location of the model image within the target image. The Point Registration tool will return the location of this match, called the *registration point*, with subpixel accuracy.

Figure 160 illustrates an example of how the Point Registration tool performs point registration.

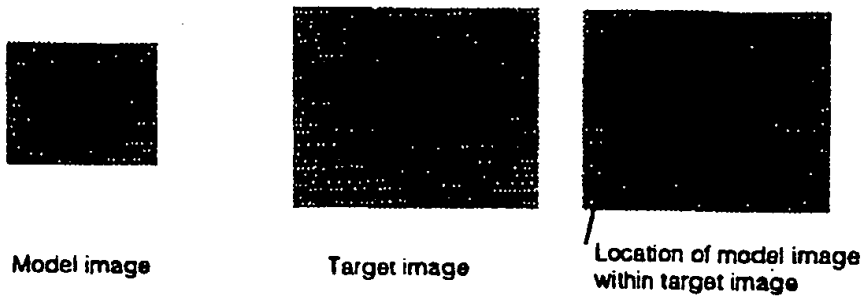


Figure 160. Point registration

6 Point Registration Tool

The Point Registration tool is optimized to locate precisely the model image within the target image, even if the target image contains image defects such as reflections, or if the model image is partially obscured by other features in the target image. Figure 161 illustrates an example of point registration where the target image is partially obscured.

Figure 161. Point registration with partially obscured target image

How the Point Registration Tool Works

The Point Registration tool finds the location of the model image within the target image. You operate the Point Registration tool by supplying the model and target images, along with the location within the target image where you expect the origin of the model image to be. The Point Registration tool will determine, with subpixel accuracy, where the origin of the model image lies within the target image.

The Point Registration tool works by computing a score indicating the degree of similarity between the model image and a particular portion of the target image that is the same size as the model image. This score can range from 0 to 255, with a score of 0 indicating that the model image and the target image are perfectly similar and a score of 255 indicating that the model image and the target image are perfectly dissimilar. The tool computes this score for locations in the immediate neighborhood surrounding the starting point. The tool will find the location within this neighborhood of the target image that produces the local minima in the value of this score.

By adding an interpolation step, the Point Registration tool then determines the location of the model image within the target image with subpixel accuracy. For typical images, the Point Registration tool can achieve accurate registration to within 0.25 pixels.

Because of the way the Point Registration tool seeks the local minima, if the starting point you specify is more than a few pixels from the actual registration point, the tool may not return the correct registration point. The exact amount of variance that the Point Registration tool can tolerate will vary depending on the images. The variance may be as small as 3 to 5 pixels for some images or as large as 30 pixels for others.

Point Registration Score

Each time it is invoked, the Point Registration tool will return, in addition to the location of the origin of the model image within the target image, a score indicating the degree of similarity between the model image and the target image. The score returned by the tool will be from 0 to 255, with a score of 0 indicating that the model image and the target image are perfectly similar and a score of 255 indicating that the model image and the target image are perfectly dissimilar.

If your point registration receives a high score, the actual precision of the point registration location may be somewhat lower than if the point registration receives a low score.

Exhaustive Point Registration

When you use the Point Registration tool, you supply the tool with two images and a starting location within the model image. The tool will confine its point registration search to a small area around the starting location that you specify.

6 Point Registration Tool

The Point Registration tool can also perform point registration exhaustively, that is, by computing the score for every possible location of the model image within the target image. This procedure, called *exhaustive point registration*, is extremely slow. It can be helpful, however, in debugging applications where the Point Registration tool does not appear to be working correctly.

Masked Point Registration

You can limit the areas of the model image and target image that the Point Registration tool uses to perform the point registration by supplying a series of rectangles to the tool. If you supply these rectangles, the tool will compute the score based only on those pixels contained within the rectangles that you specify.

Figure 162 illustrates an example of specifying a masking rectangle. In this example, the right side of the model is often obscured in the target image. By specifying a rectangle that covers the left side of the model image, you can cause the Point Registration tool to consider only the pixels in that part of the model image. This will tend to increase the accuracy of the point registration.

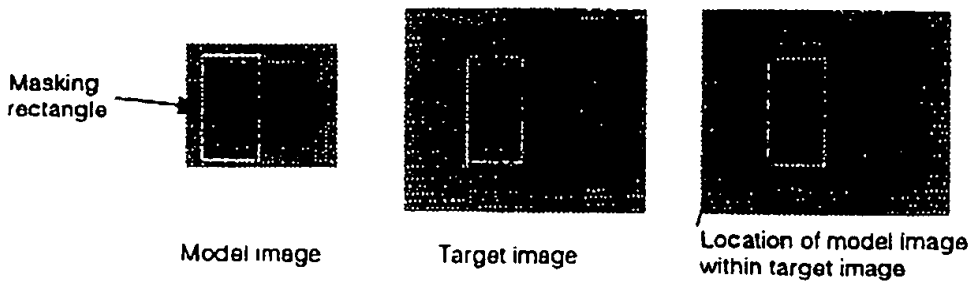


Figure 162. Masked point registration

You can also specify several masking rectangles. Figure 163 illustrates an example where the center of the model image is often obscured in the target image. By specifying a series of rectangles, you can eliminate from consideration the part of the image that is most frequently obscured.

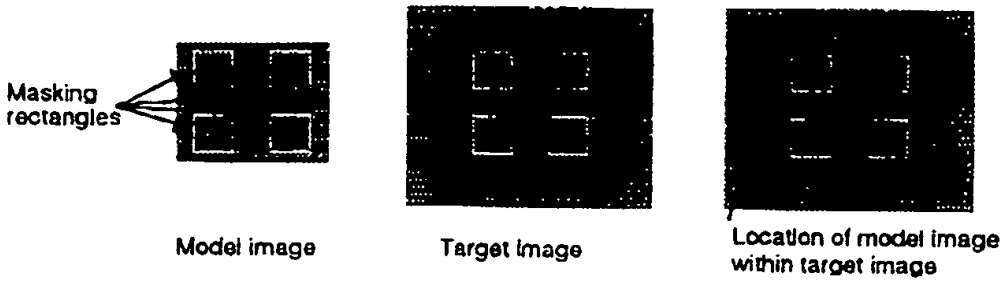


Figure 163. Masked point registration using multiple rectangles

If you specify multiple masking rectangles and the rectangles overlap, any pixels that are contained in more than one masking rectangle will be counted toward the score multiple times, once for each rectangle in which they are contained.

In all cases, the Point Registration tool will return the location of the origin of the model image within the target image.

Image Normalization

The Point Registration tool may not work well in cases where the model image and the target image have different intensity values. You can perform limited image processing as part of the point registration by supplying a pixel mapping table.

If you supply a pixel mapping table, every pixel in the target image will be mapped to the value in the pixel mapping table at the location within the pixel mapping table given by the pixel's value. For example, if a pixel in the target image had a value of 200, it would be mapped to the value of the 200th element within the pixel mapping table.

If you have specified a masking rectangle, only those portions of the target image that lie within the masking rectangle or rectangles will be mapped through the pixel mapping table.

For more information on pixel mapping, see *Chapter 3, Pixel Mapping*, in the *Image Processing* manual.

Comparing the Point Registration Tool with Search

This section compares the Point Registration tool with the Search tool. For more detailed information on the Search tool see *Chapter 1. Searching*.

The Point Registration tool has the following advantages over the Search tool:

- It is less sensitive to images that contain areas of pixels that have widely different values than the same pixels in the model image (i.e., blotches or occlusions).
- It requires no training step.

The Point Registration tool has the following disadvantages when compared with the Search tool:

- It is capable of fine alignment only.
- It does not work with scaled or rotated target images.
- It does not work well with images with brightness changes.
- It will not find or rank multiple registration points.

Using the Point Registration Tool

This section describes how to use the Point Registration tool.

Obtaining Model and Target Images

You acquire the images that you use as model and target images using the techniques described in *Chapter 1, 3000/4000 Acquisition and Display*, in the *3000/4000 Image Acquisition and I/O* manual and *Chapter 1, 5000 Acquisition and Display*, in the *5000 Image Acquisition and I/O* manual.

You should acquire all images that you plan to use with the Point Registration tool, especially images of scenes with low contrast, using a bit depth of at least 8. For fastest performance, you should acquire images as FAST8 images.

When you acquire the model image, you should take care to ensure that the image is as ideal a model as possible of the images that you will be using as target images. The Point Registration tool is designed to align target images with a wide variety of image defects, but in order for the tool to work, the model image needs to be as free from defects as possible.

Specifying a Starting Offset

When using the Point Registration tool, you must specify a starting location in the model image. The tool will perform its point registration search starting at the point that you specify. Because the tool only seeks the local minima for the score value, if you specify a starting location that varies greatly from the actual registration point, the point registration operation will fail.

You can use other vision tools such as the Search tool or the Inspection tool to locate the model feature within the target image, or you can rely on operator input to specify the coarse location of the feature.

6 Point Registration Tool

Point Registration Tool Data Structures and Functions

This section contains descriptions of the Point Registration tool data structures and functions. All Point Registration tool data structure and function names begin with the prefix **creg_**.

Structure creg_params

creg_params contains parameters that determine how the point registration search will be conducted. A structure of this type is supplied as an argument to **creg_point_register()**.

```
#include <register.h>

typedef struct
    {
    c_Int32 start_x,
        start_y,
     c_Int32 rect_count;
     char *optional_map;
     c_Int32 flags;
    } creg_params;
```

- *start_x* and *start_y* are the x- and y-coordinates of the starting position for the point registration search.
- *rect_count* is the number of rectangles in the *rects* argument to **creg_point_register()**.
- *optional_map* is an optional pixel map. If *optional_map* is non-NULL, the Point Registration tool will replace each pixel in the target image with the value contained in the element of *optional_map* corresponding to the value of the pixel in the target image. If you supply a value for *optional_map*, you must supply an image of type *FAST8*.

  You can use *optional_map* to compensate for image-to-image brightness variations.
- *flags* specifies the type of point registration that the Point Registration tool will perform. *flags* is constructed by ORing together any of the following values:
  - *CREG_TYPE1* is reserved for future use; you should always include *CREG_TYPE1* in the value of *flags*.
  - *CREG_TYPE2* is reserved for future use; you should never include *CREG_TYPE2* in the value of *flags*.
  - *CREG_NORMAL* is used to specify a point registration based on finding the local minima of the score value.

- *CREG_EXHAUSTIVE* is used to specify a point registration based on finding the global minimum of the score value.

If you specify *CREG_NORMAL* and if *start_x* and *start_y* are more than a few pixels from the actual registration point, the tool may return a location that does not represent the best match within the image. If you specify *CREG_EXHAUSTIVE*, the registration will return the best registration match for the entire image. An exhaustive point registration will be extremely slow.

Structure creg_results

creg_point_register returns a pointer to a **creg_results** structure. **creg_point_register** fills in the structure with information about the results of a point registration operation.

```
#include <register.h>

typedef struct
    {
    float x,
        y;
    c_Int32 score,
        time;
    char on_edge_x,
        on_edge_y;
} creg_results;
```

- *x* and *y* are the x-coordinate and y-coordinate, respectively, of the location within the model image at which the target image was found. *x* and *y* give the position with subpixel accuracy.
- *score* is a measure of the similarity between the pixel values in the model image and the target image at the nearest whole pixel alignment position. *score* will be between 0 and 255, with lower values indicating a greater degree of similarity between the model image and the target image.
- *time* is the total amount of time that the tool spent on this point registration, in milliseconds.
- *on_edge_x* is set to a nonzero value if, along the x-axis, one edge of the model image area is at the edge of the target image. If *on_edge_x* is nonzero, the accuracy of the position information may be slightly reduced from what it would otherwise be.
- *on_edge_y* is set to a nonzero value if, along the y-axis, one edge of the model image area is at the edge of the target image. If *on_edge_y* is nonzero, the accuracy of the position information may be slightly reduced from what it would otherwise be.

6 Point Registration Tool

Function creg_point_register()

creg_point_register() performs point registration using the model image and target image that you supply. The point registration will be controlled by the parameters in the **creg_params** structure supplied to the function. **creg_point_register()** returns a pointer to a **creg_results** structure that describes the result of the point registration.

```
#include <register.h>

creg_results *creg_point_register(const cip_buffer *target,
    const cip_buffer *model, const cia_rect *rects,
    const creg_params *params, creg_results *results);
```

- *target* points to the image to use as the target image for this point registration. *target* must be at least one pixel larger in both the x-dimension and the y-dimension than *model*.
- *model* points to the image to use as the model image for this point registration. *model* must be at least one pixel smaller in both the x-dimension and the y-dimension than *target*.
- *rects* points to an array of **cia_rect** structures. If *rects* is non-NULL, the Point Registration tool will limit the comparison of pixel values to just those pixels that lie within the rectangles contained in *rects*. If the rectangles in *rects* overlap, a pixel will be included in the comparison once for each rectangle that contains it.
- *params* points to a **creg_params** structure that defines the parameters to use for this point registration.
- *results* points to a **creg_result** structure. **creg_point_register()** will place the results of this point registration in *results* and will return a pointer to *results*.

  If you supply NULL for *results*, **creg_point_register()** will allocate a **creg_result** structure from the heap using the default allocator. You can free this structure by calling **free()**.

creg_point_register() throws *CGEN_ERR_BADARG* if

- *rects* is NULL and the *rect_count* field of *params* is not 0
- If *model* is larger than or the same size as *target*
- If the *start_x* field in *params* is less than 0 or greater than the width of *target* minus the width of *model*
- If the *start_y* field in *params* is less than 0 or greater than the height of *target* minus the height of *model*
- If *model* is NULL
- If *params* is NULL

- If *target* is NULL

6 Point Registration Tool

3 Line Finder

An Overview of the Line Finder

The Line Finder computes the locations of lines in an image. For each line that it finds, it returns an angle θ and a signed distance d from the central pixel of the image. Given θ and d, you can compute all points (x,y) in the image that satisfy the equation of the line (as illustrated in the section *The Transformation from Cartesian Space to Hough Space* on page 204). The algorithm that the Line Finder uses, along with the angle/distance classification of lines, is explained in *How the Line Finder Works* on page 166.

You can use the Line Finder in either of two modes, quick or normal. In quick mode the Line Finder provides the location of the point along each line that is nearest to the center of the image, along with the angle of the line. In normal mode, the Line Finder also estimates the length and density of each line that it locates, using statistical methods, and returns the position of the center of mass of the line.

Figure 78 illustrates the two modes of the Line Finder. The image in Figure 78a is shown in Figure 78b after the Line Finder has been invoked in quick mode. The presence of the four lines is indicated. Each line is drawn to the screen at a constant length, but no attempt is made to estimate its actual length. Figure 78c shows the result when using the Line Finder in normal mode: the length of each line segment is estimated.

For a detailed explanation of the data returned by the Line Finder in each mode, see the section *Results in Quick Mode and Normal Mode*.

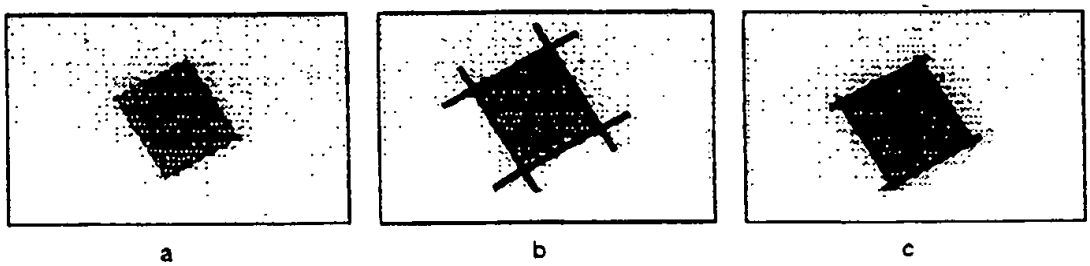


Figure 78. An image (a), lines found in quick mode (b), and line segments found in normal mode (c)

The Line Finder finds lines regardless of grey scale variations in the image, rotation of the scene, or changes in scale. A typical application for the Line Finder is locating fiducial marks in a series of images in which grey levels, scale, and rotation are unpredictable from one image to the next. Figure 79a and Figure 79b illustrate two images, each containing the same fiducial mark: a square within a square. However, the sizes of the marks, and their rotations differ between the two images. Also, the mark in Figure 79a is a dark square within

a lighter square against an even lighter background, but the mark in Figure 79b is a light square within a darker square against a darker background. As shown in Figure 79c and Figure 79d, the Line Finder finds the outlines of the fiducial marks despite these variations.

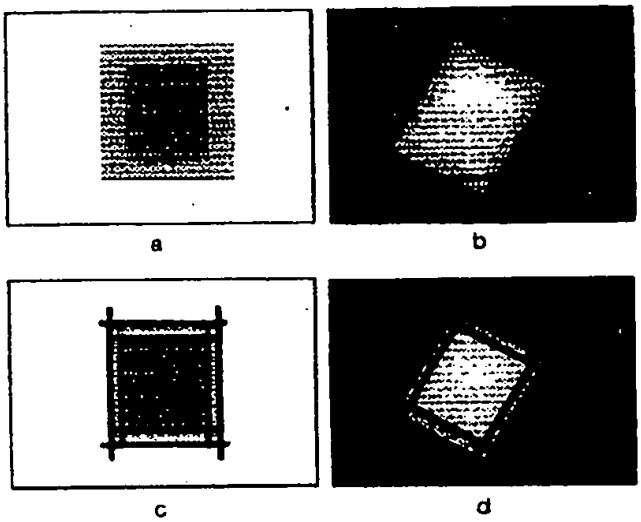


Figure 79. The Line Finder is immune to grey scale, size and rotation changes

Since the Line Finder is an intermediate-level tool, your application will probably require that you do further processing, using the results of the Line Finder as input. You will probably not be searching for lines in your image, but rather for a feature whose presence is implied by the locations of lines. For example, in Figure 79, the Line Finder does not return the locations and dimensions of the fiducial marks, only the locations of the lines that bound the nested squares. Computing the precise locations and sizes of the fiducial marks requires post-processing based on the Line Finder's results.

The Line Finder demo code includes post-processing of this sort. It contains functions that locate squares of any size and rotation, using the Line Finder. This code deduces the presence of squares based on the Line Finder's results, eliminating extraneous lines. See *Chapter 1, Cognex API Introduction*, in the *Development Environment 1* manual for the location and name of the Contour Finding demo code on your system.

3 Line Finder

How the Line Finder Works

This chapter explains how the Line Finder finds lines in an image. This discussion is divided into the following subsections:

- *Defining a Line by Angle and Distance* describes the definition of a line used by the Line Finder.
- *Hough Space* introduces the two-dimensional space in which the Line Finder records lines.
- *The Hough Line Transform* describes the algorithm that the Line Finder uses to find lines in an image.

The Line Finder uses a Cartesian coordinate system whose origin is the center of the image. Most of the examples in the book are drawn in this coordinate system, although a few use image coordinates. These two systems are pictured in Figure 80. Notice that the x-axes of the two systems are the same, but the y-axes are opposite. Note also that positive angles in Cartesian coordinates are measured counterclockwise from the x-axis; positive angles in image buffer coordinates are measured clockwise from the x-axis.

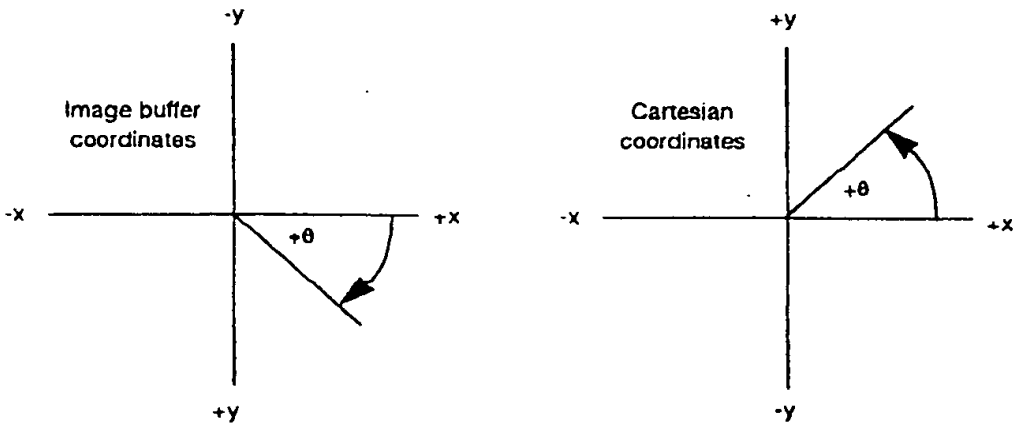


Figure 80. Image buffer coordinates and Cartesian coordinates

The Line Finder uses the central pixel of the image as a reference point. The image buffer coordinates of this pixel (x_c, y_c) are computed with integer division, using the following formulas:

$$x_c = W/2$$

$$y_c = H/2$$

where W is the width of the image and H is the height of the image, in pixels. In the Line Finder, the Cartesian coordinates of the central point in the image are (0,0).

Defining a Line by Angle and Distance

The Line Finder defines a line (in Cartesian coordinates) by its angle from the x-axis and by the signed distance from the center of the image (the central pixel) to the line. These two parameters uniquely describe any line in Cartesian space.

The feature in Figure 81a contains an edge that lies along a line (Figure 81b). The line is defined by its angle from the horizontal (θ) and by the shortest distance from the center of the image to the line (Figure 81c). Note that the distance vector is perpendicular to the line.

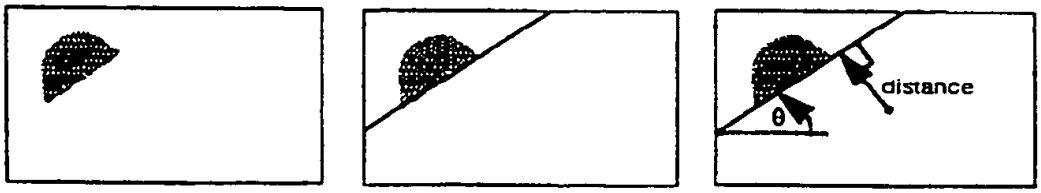


Figure 81. Angle and distance define a line

In the definition of a line by angle and distance, the distance is negative if the angle of the distance vector (in Cartesian coordinates) is greater than or equal to 0° and less than 180°; the distance is positive if the distance vector has an angle greater than or equal to 180° and less than 360°.

Distance is signed to avoid the ambiguity illustrated in Figure 82. Figure 82a shows an image with two features. Each feature is bordered on one side by a line. Although the lines are different, they have the same angle θ and the same absolute distance d from the center (Figure 82b).

3 Line Finder

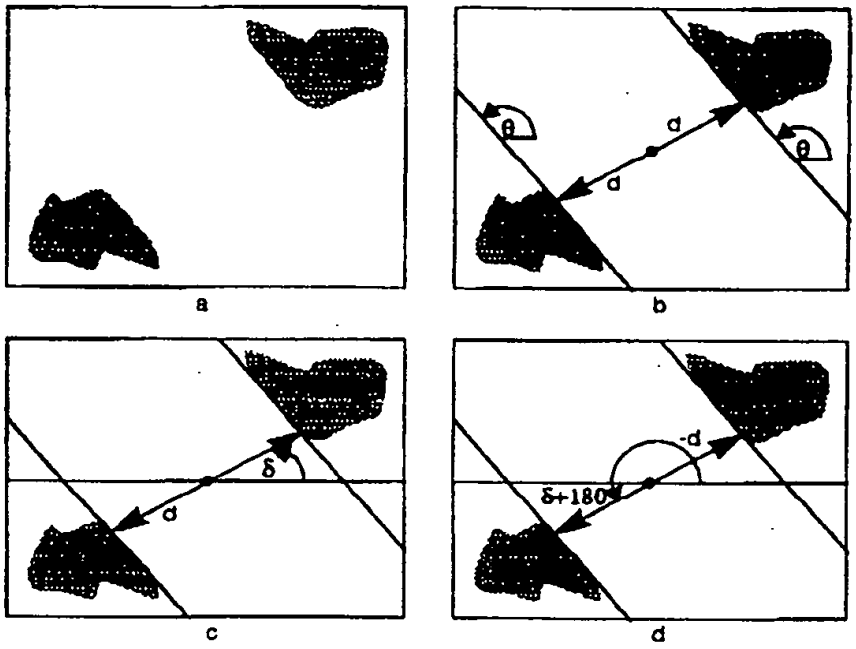


Figure 82. The angle of the distance vector determines the sign of the distance

By designating a sign to the distance vector, based on its angle (δ in Figure 82c, δ+180 in Figure 82d), each line is ensured a unique definition. In this example, the two lines are defined as (θ, *d*) and (θ, -*d*).

Once you know the distance and the angle of a line, you can compute all points (x_0, y_0), in Cartesian coordinates, that lie on the line. Given a distance, *d* and an angle θ, a line is all points (x_0, y_0) that satisfy the following equation:

$$d = x_0 \sin\theta - y_0 \cos\theta$$

The derivation of this formula is supplied in the section *The Transformation from Cartesian Space to Hough Space* on page 204. As an example of its application, if θ is 45 and *d* is 0, a line is all points in Cartesian space that satisfy the following equalities:

$$0 = x\sin(45) - y\cos(45)$$

$$0 = x\frac{1}{\sqrt{2}} - y\frac{1}{\sqrt{2}}$$

$$x = y$$

Hough Space

Hough space is a two-dimensional space in which the Line Finder records the lines that it finds in an image. A point in Hough space represents a line; one axis represents angle, the other represents distance.

Hough space is implemented in the Line Finder as an array: a simple example is shown in Figure 83. The Hough space in this example can record eight angles and nine distances and therefore 72 lines.

The Line Finder lets you control the size of Hough space. You specify the range of distance in pixels, and, in the edge detection data structure that you pass to the Line Finder, you supply the number of angles that the Line Finder can compute.

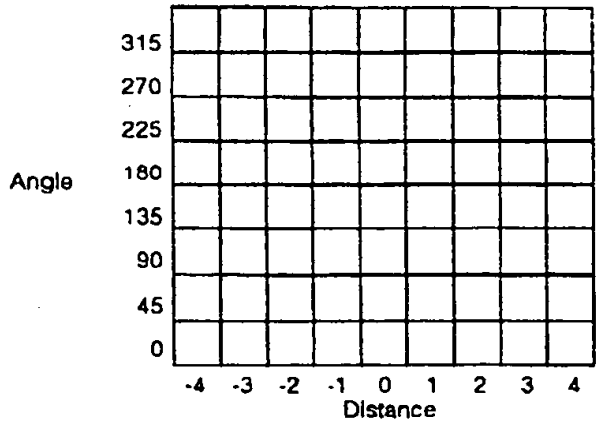


Figure 83. Hough space

The Hough space in Figure 84 contains a single line. It has an angle of 135° and is a distance of 3 units from the center of the image.

3 Line Finder

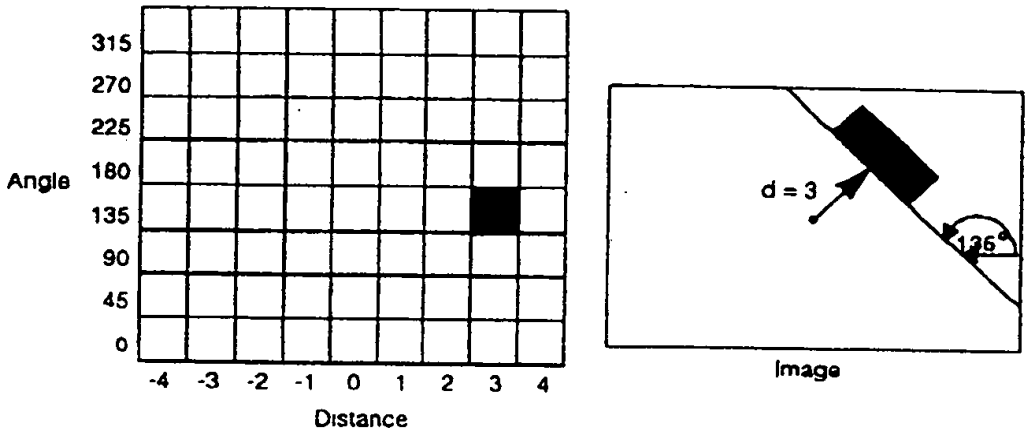


Figure 84. Hough space containing a single line

The Hough Line Transform

The Line Finder uses the Hough Line Transform to locate lines in an image. This method is outlined and illustrated in this section. To understand this discussion, you should be familiar with edge detection and peak detection, as described in *Chapter 4, Edge Detection Tool*, in the *Image Processing* manual.

The Hough Line Transform, as implemented by this tool, takes advantage of the following rule: If an edge pixel's angle ϕ and location $(x0,y0)$ is known, then the line on which the pixel lies is also known; it is the line with angle $\phi+90$ that contains the pixel $(x0,y0)$. This is illustrated in Figure 85.

Line Finder 3

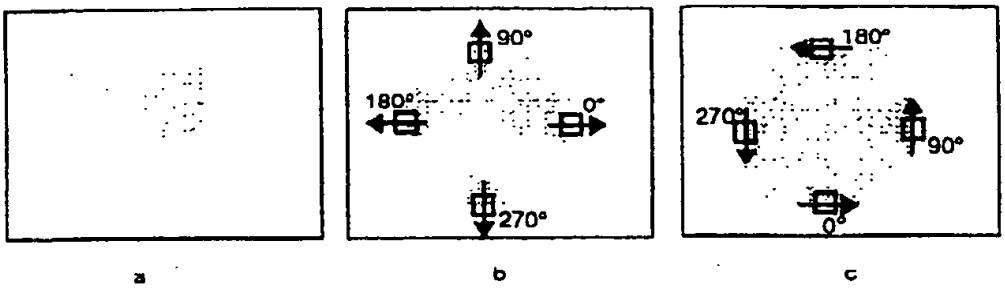


Figure 85. Four edges in an image (a), edge angles at four edge pixels (b), and line angles at those pixels (c)

The Hough Line Transform is implemented as follows:

1. Using edge detection, the Line Finder creates an edge magnitude image and an edge angle image from the input image.
2. The Line Finder creates and clears a Hough space.
3. For each edge pixel in the input image, the Line Finder does the following:
   - Using the edge angle ϕ and location $(x0,y0)$ of the pixel, the Line Finder calculates the line's distance from the center, d, and its angle, θ (using the formula $\theta = \phi + 90$).
   - The Line Finder increments the bin in Hough space representing the line (θ,d).
4. Once all edge pixels in the image have been examined, the Line Finder searches Hough space for maximum values, using peak detection. The highest value in Hough space represents the strongest line in the image, the second highest represents the second strongest line, and so forth.
5. The location of each line is printed out and, optionally, each line is drawn in an output image.

Figure 86 through Figure 89 illustrate this method.

Figure 86a is an image passed to the Line Finder. Figure 86b is a stylized edge magnitude image created from the input image.

3 Line Finder

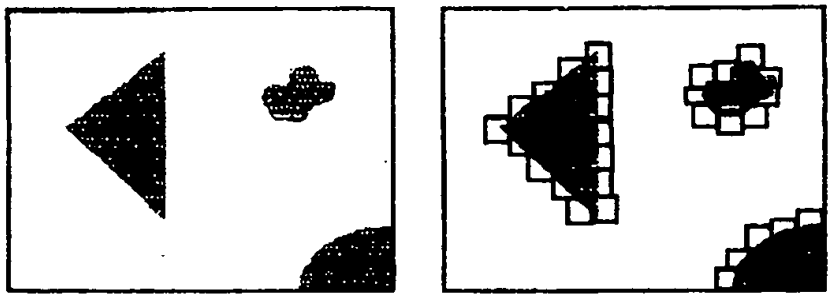


Figure 86. An input image (a) and the edge detected input image (b)

Each edge pixel is processed, as described above. For example, because the outlined edge pixel in Figure 87 belongs to the line (90, -2), the bin in Hough space that represents that line is incremented.

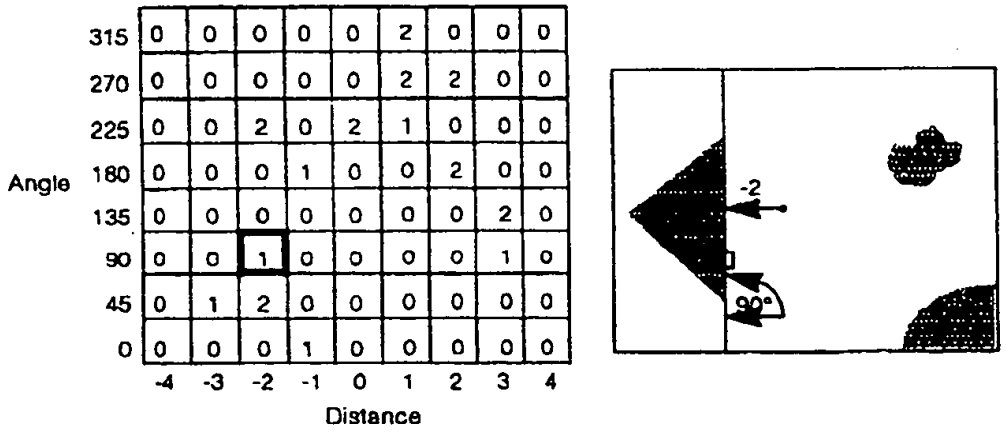


Figure 87. The bin in Hough space for the edge pixel's line is incremented.

Figure 88 shows the final Hough space along with the input image. Figure 89 shows the peak-detected Hough space, with each of the peaks outlined. Figure 89 also shows the input image with the lines, represented by the peaks in Hough space, drawn into the input image.

Angle	-4	-3	-2	-1	0	1	2	3	4
315	0	4	0	0	0	2	0	0	0
270	0	1	0	0	0	2	2	3	0
225	0	0	2	0	2	1	0	0	0
180	0	0	0	1	0	0	2	1	0
135	0	0	0	0	0	0	0	2	0
90	0	0	7	0	0	0	0	1	0
45	0	1	2	0	0	0	0	1	0
0	0	0	1	1	0	0	0	0	0

Distance

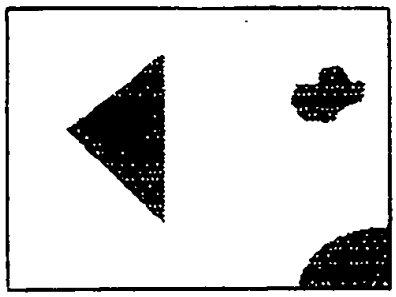


Figure 88. The final Hough space and the input image

3 Line Finder

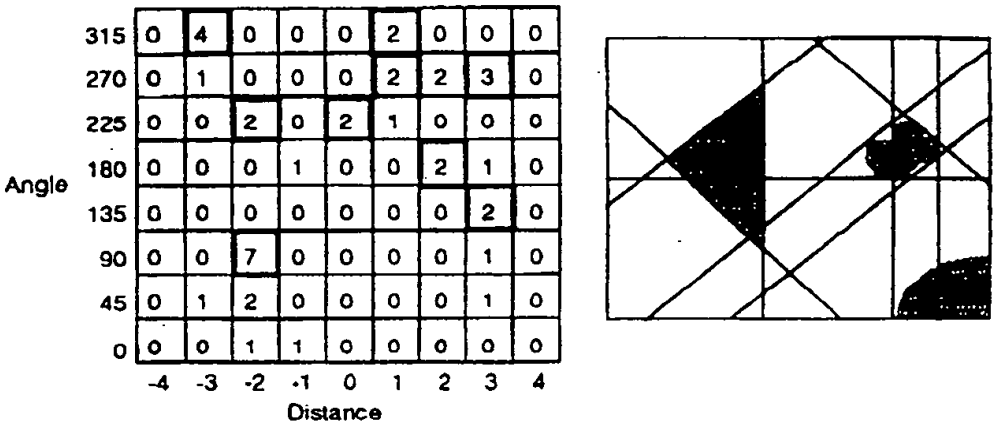


Figure 89. The peak-detected Hough space and the input image, with lines added

Notice that the Hough space contains many extraneous lines. If, for instance, the vision problem is to find the triangle in the image, the lower scoring lines that border the blob need to be eliminated. Almost all of the Line Finder applications require post-processing of this sort. In this example, simply locating the three highest peaks in Hough space eliminates the unwanted lines (Figure 90). In most applications, the required post-processing is more complex.

Angle	-4	-3	-2	-1	0	1	2	3	4
315	0	4	0	0	0	2	0	0	0
270	0	1	0	0	0	2	2	3	0
225	0	0	2	0	2	1	0	0	0
180	0	0	0	1	0	0	2	1	0
135	0	0	0	0	0	0	0	2	0
90	0	0	7	0	0	0	0	1	0
45	0	1	2	0	0	0	0	1	0
0	0	0	1	1	0	0	0	0	0

Distance

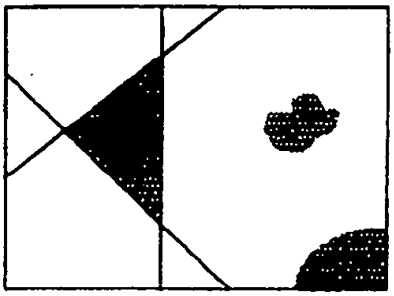


Figure 90. The output image after post-processing

APPENDIX II

Patent Application for

MACHINE VISION CALIBRATION TARGETS AND METHODS OF DETERMINING THEIR LOCATION AND ORIENTATION

Software Listings

THE FOLLOWING APPENDIX IS NOT BELIEVED TO BE NECESSARY FOR ENABLEMENT OR BEST MODE DISCLOSURE OF THE INVENTION DISCLOSED AND CLAIMED IN THE ACCOMPANYING APPLICATION.

```
/*
 * Copyright (c) 1996 by Cognex Corporation, Natick, MA USA
 * All rights reserved. This material contains unpublished,
 * copyrighted work, which includes confidential and proprietary
 * information of Cognex.
 *
 *   cwa_find_target.c, Aaron Wallack - 13 Apr 1996: Created.
 *   Time-stamp: <96/07/30 09:32:47 awallack>: last modified.
 *
 * $Revision: /main/none$
 */

#include <prealign.h>
#include <math.h>
#include <stdio.h>
#include <cct.h>
#include <cip.h>
#include <ctm.h>
#include <caq/caq_def.h>
#include <cwa_priv.h>
#include <search/ctr_def.h>
#include <search/cse_def.h>
#include <clp/cclp.h>
#include <clp/cclp_er.h>
#include <chp.h>

/* This file includes a function for localizing a crosshair (two
 * square) target.

 * ############
 * ############
 * ############
 * ############
 * ############
 * ############
 * ############
 *              ###############
 *              ###############
 *              ###############
 *              ###############
 *              ###############
 *              ###############
 *              ###############
 *

 * The algorithm is intended to handle the cases when the target is
 * imperfect (occluded or broken)

 * ############################
 * ############################ (occlusion)
 * ############
 * ############
 * ############
 * ############
```

```
* ############
*              ################
*              ################
*              ################
*              ################
*       broken  ##############
*                ##############
*              ################

* These routines are intended to be used as a subroutine for
* determining the cameras' field of views (with respect to an absolute
* coordinate system defined by a motion stage).

* The target localization routine (cwa_find_target) takes as input:
* img - a cip_buffer containing an image of the target
* calib - a calibration structure intended to characterize the
*         non-rectangularity of the pixels
* angle - the orientation of the crosshair in terms of image
*         coordinates (we require this
*         orientation estimate to be accurate to 3 degrees, 0.05 radians
* result - pointer to a cwa_line structure where the result will be
*          returned

* Requirements:

* Image quality:

* The image of the crosshair must be visible and perfect
* (without occlusion or breaks) throughout a 100 X 100 pixel square
* surrounding the crosshair center.
* Consequently, the center of the crosshair must be at least
* 50 pixels away from the borders of the cip_buffer.

* Orientation estimate accuracy:
* The given angle estimate (given in terms of image coordinates) must be
* accurate to within 3 degrees 0.05 radians

* This algoritm  tolerates occlusion and breaks in the target
* so long as these imperfections occur at least 50
* pixels away from the crosshair center

* Major Consideration:

* One interesting thing to note about crosshair calibration, and the reason
* that we use four distinct rays, is that the separation between the opposite
* rays depends upon the lighting and aperture of the lens. Furthermore, we
* cannot assume that the separation between these edges will remain constant
* from one setup to another

* What we really want is the position of the '+'

* ############
* ############
* ############
```

```
*  ############
*  ############
*  ############
*  ############
*                +
*                 ################
*                 ################
*                 ################
*                 ################
*                 ################
*                 ################
*
*
* Since we do not know what the offset/separation is, it turns out that the
* repeatability of the entire fit (localization of the crosshair) is bounded
* by the accuracy of the smallest of these four lines. It doesn't make any
* sense to include 10000 edge points from one ray if we only have 50 edge
* points from another (the localization error will still be on the order of
* 0.1/sqrt(50) pels, assuming random i.i.d. error and subpixel localization
* accuracy of 0.1 pixel).

* We want to incorporate the same number of edge points from opposite rays
* because we always want to fit the lines to the same exact points 'for
* highest accuracy and repeatability'. When the calibration target is
* occluded by a particular object (such as the borders that can be seen in
* the database), then if we use the exact same number of pixels on the
* opposite unoccluded ray as the number of pixels on the occluded ray, then
* we will always be looking at the same pixels on the unoccluded ray

* Basically, since we do not know which edge points correspond to
* imperfections in the target and which edge points correspond to
* valid points on the target, we use a boot-strapping algorithm to
* compute regions where edge points are VALID.

* Assuming that the target is perfect within the GUARANTEED GOOD
* REGION (center +/- 50 pixels), we can compute four GOOD RAYS
* characterizing the four rays eminating from the crosshair center.
* Then, we can use these GOOD RAYS to define the VALID REGIONS (the
* set of points within a TIGHT_THRESHOLD of the GOOD RAYS). Finally,
* we improve the estimates of the four rays by going back and
* enumerating edge points along the GOOD RAYS until we encounter an
* edge point which falls outside the VALID REGIONS.

* Definitions
*
* GUARANTEED GOOD REGION
* region surrounding the origin where we insist that the calibration target
* must be perfect

* GOOD RAY
* A ray fit to points enumerated within the guaranteed good region

* VALID REGION
```

```
* Region surounding good ray where we trust that the edge points are due to
* the crosshair and not from occlusion or breaks

* OUTLIER POINTS
* Points which we cannot trust because they do not fall within the valid
* region

*   CENTER OF CROSSHAIR TARGET
*         V                                                    outlier
*         V                                                  edge points
*         ###########################################################*******
*-------------------------------------------------------------------
*         ##################################################
*         ########################################
* ########  <-----------good ray-------------------------------------------->
* ########
* ########          VALID REGION
*-------------------------------------------------------------------
* ########

*         |<--------guaranteed good region------->

* We are adopting a better-safe-than-sorry strategy for handling
* occlusion (as soon as we see a "bad" point, we lose faith in all of
* the subsequent points). Although this approach is suboptimal, the
* database images seem to contain nearly perfect points eminating
* from the origin that it should work well in practice.

* Algorithm:

* 1) Use absolute normalized correlation search to approximately
*    locate the crosshair center
*    This involves constructing a synthetic model of a rotated
*    crosshair center (see gross_locate_crosshair_center_in_image())

* 2) Given this initial position estimate and the user-supplied orientation
* estimate, improve the positional and orientational estimates by localizing
* four good rays eminating from the crosshair center.  This involves
* enumerating edge points within the GUARANTEED GOOD REGION (50 pixels)
* along the supplied orientation as well as
* theta+PI/2,theta+PI,theta+3*PI/2.  These computed four GOOD RAYS are used
* in step 3 to check whether edge points are outliers. In other words, these
* four good rays define VALID REGIONS separating good edge points (points we
* believe correspond to the crosshair) from bad edge points (points we
* believe are due to occlusion or breaks) We should also point out that we
* fix the orientations of the four rays by "averaging" these orientations,
* i.e., we incorporate the constraint that the four rays correspond to
* orthogonal physical lines.  (see
* compute_center_position_and_angle_assuming_four_guaranteed_good_rays())

* 3) Go back and enmerate edge points along the four rays until we find an
* edge point which does not satisfy the VALID REGION constraints.  We also
* throw out some of the edge points in order to realize equal numbers of
* edge points from opposite rays.  (see
```

```
* compute_center_position_and_angle_along_guaranteed_good_rays();

* NOTES
* -----

* When we enumerate edge points along rays eminating from the origin,
* we start out a small distance away from the origin so that we don't
* get confused by two different edges (this buffer zone is
* characterized by the constant CIRCULAR_BUFFER_ZONE_AROUND_CENTER,
* which is 10 pixels)

* The central function in this algorithm is
* enumerate_points_and_fit_line_along_ray() which enumerates edge points
* along a specified path. enumerate_points_and_fit_line_along_ray() applies
* calipers at sample points along the path (the edge estimates can be
* further improved using parabolic subpixel interpolation) Along a specific
* path, enumerate_points_and_fit_line_along_ray() always makes integral unit
* steps in either the x or y direction. This is so that we can apply
* axis-aligned calipers along every row or column when the step size is 1

* enumerate_points_and_fit_line_along_ray() enumerates edge points
* until it either encounters a bad edge point, or finishes sampling
* the edge segment. At this point, it ceases enumerating edge points,
* and fits a line to the points. There are four ways an edge point
* can be "bad". If enumerate_points_and_fit_line_along_ray()
* encounters any of these four situations, it returns 0 after fitting
* the line; if it successfully enumerated all of the edge points
* along its specified course, it returns 1.

1. no edge is found by the caliper applied at the sample point

2. more than one edge is found by the caliper and the highest
   scoring edge is less than twice the score of the second highest
   scoring edge (this 2X comes from the CONFUSION_THRESHOLD)

3. the computed edge point is an outlier (the distance between the
   edge point and the nominal line is larger than threshold)

4. the caliper extends outside of the cip_buffer

* When we call enumerate_points_and_fit_line_along_ray() in step 1, we
* specify a ray from a point near the crosshair center to a point 50
* pixels away from the crosshair center. We do look at the return val
* to check that the enumerate_points_and_fit_line_along_ray() did not
* encounter any problems.

* When we call enumerate_points_and_fit_line_along_ray() in step 2, we
* specify a ray from a point close to the crosshair center along a
* good ray to a point 1000 pixels away (guaranteed to be outside the
* cip_buffer). Thereby, even if we do not see any "bad" points, we
*  will undoubtedly leave the cip_buffer at some point.

* The reason we pass a calibration object to cwa_find_target is that we
* recompute the orientations of the four rays by incorporating the knowledge
```

```
* that they all come from orthogonal lines. If the pixels are non-square
* (i.e., by virtue of the fact that the CCD elements are non-square), then
* we need to account for this when we "average" the four orientations. We
* "average" the four orientations by transforming the image orientations
* into physical coordinates (and then truly averaging the physical angles)
* and then transforming the average angle back into image coordinates.  We
* try to use the term "phys" to denote physical coordinates and "img" to
* denote image coordinates (such as phys_orient and img_orient).

* We use different sized calipers for steps 2 & 3 because we make
* different assumptions about the relative accuracies of our caliper
* application points. We use the terms LOOSE and TIGHT to refer to the
* calipers; LOOSE for step 2 where we have little faith in the
* application points, and are thereby forced to use broad calipers, and
* TIGHT for step 3, where we have much more faith in the application
* points, and can thereby use relatively tight calipers.

* We expect the initial search position to be accurate to 2 pels, and
* the orientation estimate to be accurate to 0.05 radians. At a caliper
* application point 50 pels away, the caliper may be off by up to 5
* pels from the correct edge position. Just to be safe, we use a
* caliper with projection length 24 (12 on each side) to enumerate edge
* points in step 2

* After improving the position and orientation estimates in step 2, we
* basically assume that we've got the right position to - - 0.2 pels
* and +/- 0.01 radians. Thereby, we can use these lines to threshold
* outliers because we would expect errors of only 2.2 pels at 200
* pixels away. This is pretty tight because we use a threshold of 3.0
* pels to detemine outliers

* (actually, the 0.2 pels, 0.01 radians could significantly impact the
* performance because the first 50 pels and the orientation estimate
* end up predominantly affecting which edge points are included in the
* final line estimates)

* Main subroutines -

* gross_locate_crosshair_center_in_image()
* construct a synthetic model of the crosshair and use search to
* estimate the crosshair position

* fit_line_to_points()
* fit a line (x,y,t) to an array of points

* compute_subpixel_position()
* use parabolic interpolation at the neighborhood of the application point
* to compute the subpixel edge position

* enumerate_points_and_fit_line_along_ray() enumerate edge points along a
* line segment from start_point to end_point and possibly store the found
* edge points in an array.  Furthermore, a nominal line and a threshold are
* passed in as arguments, and when the found edge points deviate from this
* nominal line by more than the threshold, enumeration is stopped and a line
```

```
 * is fit to the enumerated edge positions. Enumeration is also stopped when
 * two edges are found (confusion), no edges are found (occlusion) or when
 * the caliper is applied out of bounds (out of bounds)

 * average_four_image_orientations()
 * average the orientations of the four rays to compute the best
 * composite orientation estimate. This function transforms the image
 * orientations into the physical domain in order to exploit the
 * constraint that the physical lines are orthogonal

 * compute_center_position_and_angle_assuming_four_guaranteed_good_rays()
 * improve estimate of the crosshair position and orientation by
 * searching four regions of guaranteed good length (50 pixels)
 * eminating from the crosshair center. Lines are fit to the edge
 * points along the four regions, and then the orientations of these
 * lines are "averaged". Two orthogonal lines are computed by
 * averaging pairs of the four rays Finally, the position estimate is
 * computed by intersecting the two orthogonal lines

 * compute_center_position_and_angle_along_guaranteed_good_rays() generate
 * final estimate of crosshair position and orientation by enumerating points
 * along four rays (using four good rays as thresholding lines). We use
 * tighter calipers (than we used in step 2) because we trust the positions
 * of the good rays.
 */

#define N_EDGES 10
#define GUARANTEED_GOOD_SIZE 50
#define GROSS_TARGET_SIZE GUARANTEED_GOOD_SIZE
#define NUM_SAMPLE_POINTS 1000
#define LOOSE_THRESHOLD 12.
#define TIGHT_THRESHOLD 3.
#define LOOSE_CALIPER_WIDTH 24
/* ((int)2*(LOOSE_THRESHOLD+GUARANTEED_GOOD_SIZE*sin(0.05)) + 2 (kernel)*/
#define TIGHT_CALIPER_WIDTH 14
/* ((int)3*TIGHT_THRESHOLD) + 2 (kernel)*/

/*
 * TIGHT_CALIPER_WIDTH should be larger than the minimum width so that
 * we check whether there is a sharp edge just outside the
 * TIGHT_THRESHOLD range, because if there is a sharp edge just
 * outside the TIGHT_THRESHOLD range, then we want to declare
 * confusion and stop enumerating points
 */

#define LOOSE_CALIPER_PROJECTION_LENGTH 5
#define TIGHT_CALIPER_PROJECTION_LENGTH 5
#define MAX_NUM_POINTS 1000
#define EPSILON_THRESHOLD 0.05
#define CIRCULAR_BUFFER_ZONE_AROUND_CENTER (8.+LOOSE_CALIPER_PROJECTION_LENGTH)
#define STEP_SIZE_USED_TO_ENUMERATE_POINTS_IN_GOOD_REGION 3
#define STEP_SIZE_USED_TO_ENUMERATE_ALL_POINTS 1
#define MAX_LENGTH_DIAGONAL_ACROSS_CIP_BUFFER 1000.
#define CONFUSION_THRESHOLD 3.
```

```
/* number of pixels to ignore when we run into an outlier */
#define OUTLIER_BUFFER 7
extern int cd_showas(int, char*);
extern double cwa_point_distance();

char* cy_prealign_cwa_find_target() { return __FILE__; }

static int cwa_find_target_debug_flag=0;

int set_cwa_find_target_debug(int x)
{
    int old_flag=cwa_find_target_debug_flag;
    cwa_find_target_debug_flag=x;
    return old_flag;
}

typedef enum calib_orient
{
    CALIB_HORIZ=0,
    CALIB_VERT
} calib_orient;

/* A training parameters record for training rotated crosshair models */
static ctr_params cwa_calib_corr_tp =
  { 2,              /* make a binary model (2 grey levels)          */
    0,              /* no leniency                                  */
    0,              /* don't measure angle                          */
    5,              /* default bias                                 */
    1,              /* left tail at 1%                              */
    1,              /* right tail at 1%                             */
    50,             /* threshold, not used for grey models          */
    4, 4,    /* model resolution                                    */
    0,              /* don't train for reader                       */
    0,              /* no angle training                            */
    0,              /* ditto                                        */
    0.0,     /* no standard deviation thresholding for statistics   */
    128,     /* care threshold for mask training                    */
      0             /* NO flags set                                 */
           /* (do not bother to initialized reserved fields)        */
  };

static cse_params
cwa_calib_sp = {cse_absolute, cse_abs_binary, NO, NO, 1, 150, 1000};

/* caliper used to find more accurate position estimates of edge
 * positions
 */

static
cclp_constraint cwa_find_target_gcs[1] =
{
    {CCLP_CONTRAST,     0,    1000, 0,    100,  0},
};
```

```
/* static */
cclp_caliper clp_find_target_initial_search = {
    LOOSE_CALIPER_WIDTH,/* Search Length  */
    LOOSE_CALIPER_PROJECTION_LENGTH,/* Projection Length     */
    2,                  /* Edge Filter Size      */
    0,                  /* Edge Filter Leniency */
    10.,          /* Expected Size  */
    EDGE_THRESHOLD,     /* Contrast threshold    */
    CCLP_DONT_CARE,     /* E1 Polarity           */
    CCLP_NO_PAIRS,      /* E2 Polarity             */
    CCLP_DEFAULT,       /* Optimization          */
    8,                  /* Bits per pixel */
    TRUE,         /* Window Rotation       */
    TRUE,         /* Interpolate           */
    1,                  /* Number of contraints */
    cwa_find_target_gcs,/* Constraints           */
    NULL,         /* Pixel Map             */
    0, 0, 0
};

/* static */
cclp_caliper clp_find_target_fine_search = {
    TIGHT_CALIPER_WIDTH,/* Search Length  */
    TIGHT_CALIPER_PROJECTION_LENGTH,/* Projection Length     */
    2,                  /* Edge Filter Size      */
    0,                  /* Edge Filter Leniency */
    10.,          /* Expected Size  */
    EDGE_THRESHOLD,     /* Contrast threshold    */
    CCLP_DONT_CARE,     /* E1 Polarity           */
    CCLP_NO_PAIRS,      /* E2 Polarity             */
    CCLP_DEFAULT,       /* Optimization          */
    8,                  /* Bits per pixel */
    TRUE,         /* Window Rotation       */
    TRUE,         /* Interpolate           */
    1,                  /* Number of contraints */
    cwa_find_target_gcs,/* Constraints           */
    NULL,         /* Pixel Map             */
    0, 0, 0
};


/* static */
cclp_caliper clp_find_target_compute_subpixel = {
    TIGHT_CALIPER_WIDTH,/* Search Length  */
    1,/* Projection Length     */
    2,                  /* Edge Filter Size      */
    0,                  /* Edge Filter Leniency */
    10.,          /* Expected Size  */
    EDGE_THRESHOLD,     /* Contrast threshold    */
    CCLP_DONT_CARE,     /* E1 Polarity           */
    CCLP_NO_PAIRS,      /* E2 Polarity             */
    CCLP_DEFAULT,       /* Optimization          */
    8,                  /* Bits per pixel */
    FALSE,        /* Window Rotation       */
```

```
    FALSE,          /* Interpolate          */
    1,                  /* Number of contraints */
    cwa_find_target_gcs,/* Constraints          */
    NULL,           /* Pixel Map            */
    0, 0, 0
};

/* make an image of a rotated center of a crosshair. this image is
 * used to train a model to search for the crosshair in an image
 * img - cip_buffer where rotated crosshair will be stored
 * orientation_in_tenths - orientation of rotated crosshair in tenths
 *                         of degrees
 * returns img
 */
cip_buffer *make_cross_hair_center_image
(cip_buffer *img, int orientation_in_tenths)
{
    cip_buffer win, *xhair=NULL;
    cct_signal sig;

    NO_REGISTER(xhair);

    if (!img)
      cct_error(CGEN_ERR_BADARG);

    if (sig=cct_catch(0))
      goto done;

    /* allocate a temporary cip_buffer (xhair) which will contain a
     * non-rotated crosshairs, and then use this temporary cip_buffer
     * as a source for cip_rotate_centered
     */
    xhair = cip_create(img->width*2,img->height*2,8 );

    cip_set(xhair,0);

    cip_window(xhair,&win,0,0,xhair->width/2,xhair->height/2);
    cip_set(&win,255);
    cip_window(xhair,&win,xhair->width/2,xhair->height/2,
            xhair->width/2,xhair->height/2);
    cip_set(&win,255);

    cip_rotate_centered(xhair,img,orientation_in_tenths);

done:
    if (xhair) cip_delete(xhair), xhair=NULL;
    if (sig) cct_throw(sig);

    return img;
}

/* construct a synthetic image and use search to find a gross estimate
 * of the center position of the crosshair center
 * we expect that the synthetic model will be accurate to
```

```
 * approximately 1 pixel, and therefore, this search routine should
 * localize the center to approximately 1 pixel

 * img - image containing a rotated crosshair
 * orientation_in_tenths - orientation of rotated crosshair in tenths
 *                         of degrees
 * res - pointer to cwa_point where result will be stored

 * gross_locate_crosshair_center_in_image () returns int signifying
 * whether or not it found a crosshair
 */
static int gross_locate_crosshair_center_in_image
(cip_buffer *img, int orientation_in_tenths, cwa_point *res)
{
    cip_buffer *target=NULL;
    cse_model *mdl=NULL;
    cct_signal sig;
    cse_results cse_res;
    int ans;

    NO_REGISTER(target);
    NO_REGISTER(mdl);

    if (sig=cct_catch(0))
      goto done;

    target = cip_create(GROSS_TARGET_SIZE,GROSS_TARGET_SIZE,8);

    make_cross_hair_center_image(target,orientation_in_tenths);

    mdl = ctr_train_model(target, 0, 0, 0, 0, 0, 0, &cwa_calib_corr_tp  0);

    cse_area_search(img, mdl, &cwa_calib_sp,&cse_res);
    if (cwa_find_target_debug_flag & 0x1)
      printf("%d %d %d %d\n",
             cse_res.x,cse_res.y,cse_res.score,cse_res.found);

    res->x = 1.*(GROSS_TARGET_SIZE/2 + CIA_RND16(cse_res.x.n));
    res->y = 1.*(GROSS_TARGET_SIZE/2 + CIA_RND16(cse_res.y.n));
done:
    if (target) cip_delete(target), target = NULL;
    if (mdl) ctr_delete_model(mdl), mdl=NULL;
    if (sig) cct_throw(sig);

    return cse_res.found;
}

/* compute distance from a point to the closest point on a line */
static double distance_from_point_to_line(cwa_point *p1, cwa_line *l1)
{
    cwa_line l2;
    cwa_point p2;

    l2.x = p1->x;
```

```
    l2.y = p1->y;
    l2.t = l1->t + PI/2;
    cwa_lines_to_point(l1,&l2,&p2);
    return (sqrt((p1->x-p2.x)*(p1->x-p2.x) +
             (p1->y-p2.y)*(p1->y-p2.y)));
}

/* Fit a line to an array of points */
static double fit_line_to_points
(cwa_point *pts, int n_points, cwa_line *line)
{
    int j;
    double xsum=0.0, ysum=0.0, xcent, ycent;
    double a, b, c, theta, xp, yp;
    cwa_point *ptr;

    if (n_points < 2)
      cct_error(CWA_ERR_NO_INTERSECT);

    for (j=0, ptr = pts; j<n_points; j++, ptr++) {
      if (cwa_find_target_debug_flag & 0x1)
          printf("%f %f\n",ptr->x,ptr->y);
      xsum += ptr->x;
      ysum += ptr->y;
    }

    xcent = xsum / n_points;
    ycent = ysum / n_points;

    for (j=0, ptr = pts, a=b=c=0.; j<n_points; j++, ptr++) {
      xp = ptr->x - xcent;
      yp = ptr->y - ycent;
      a += xp * xp;
      b += 2 * xp * yp;
      c += yp * yp;
    }

    theta = atan2(b, a-c) / 2.0;

    line->x = xcent;
    line->y = ycent;
    line->t = theta;

    if (cwa_find_target_debug_flag & 0x1){
      printf("fit_line_to_points returned: %f",
              (a*cos(theta+PI/2)*cos(theta+PI/2)+
             b*cos(theta+PI/2)*sin(theta+PI/2)+
             c*sin(theta+PI/2)*sin(theta+PI/2))/n_points);
      cd_showas((int)line,"cwa_line");
    }
    return ((a*cos(theta+PI/2)*cos(theta+PI/2)+
           b*cos(theta+PI/2)*sin(theta+PI/2)+
           c*sin(theta+PI/2)*sin(theta+PI/2))/n_points);
}
```

```
static int print_list_of_points
(cwa_point *val, int num_vals)
{
    int i;
    for (i = 0; i < num_vals; i++)
      printf("%f %f\n",val[i].x,val[i].y);
}

/* calib_orient_from_angle() returns CALIB_VERT or CALIB_HORIZ
 * depending upon the orientation (measured in degrees) of a line. Its
 * purpose is to determine whether we need to use a vertical or
 * horizontal quadratic subpixel edge interpolation
 */
static calib_orient calib_orient_from_angle(double angle_in_degrees)
{
    double angle_in_rad;
    angle_in_rad = angle_in_degrees*PI/180.;

    if (fabs(cos(angle_in_rad)) >
      fabs(sin(angle_in_rad)))
      return CALIB_VERT;
    return CALIB_HORIZ;
}

/* computes subpixel edge positions given an image and a neighborhood
 * (ap_x, ap_y), to search for an edge. Depending upon the
 * orientation_in_degrees, compute_subpixel_position() will either
 * perform a vertical or horizontal subpixel edge estimation. Subpixel
 * edge positions are computed in the same manner as the boundary
 * tracker (using 7 neighboring pixels, and 7 differences  where
 * we quadratically interpolate the edge position from at least three
 * differences.

 * We compute a seven first differences just to be safe. There is no
 * guarantee that the computed caliper position corresponds to the maximum
 * 1st difference (because we're using calipers with filter size 2)

 * img - cip_buffer containing image of crosshair target
 * ap_x,ap_y - application point where we sample image grey values
 *             actually, we sample grey values at the application
 *             point and +/- 2 from the application point (maybe we
 *             should sample at +/- 4, who knows
 * orientation_in_degrees - orientation of caliper which has been applied
 *                          to find this application point. Since we
 *                          only want to call subpixel interpolation
 *                          at the right place, we suggest using
 *                          on-axis calipers
 * pos_x, pos_y - pointers to doubles where
 *                compute_subpixel_position() will store result
 */
int compute_subpixel_position
(cip_buffer *img, int ap_x, int ap_y,
 double orientation_in_degrees, double *pos_x, double *pos_y)
```

```
{
    int p0,p1,p2,p3,p4,p5,p6,p7;
    int d0,d1,d2,d3,d4,d5,d6, d_max, d_min;
    calib_orient direction;

    if (cwa_find_target_debug_flag & 0x1)
      printf("compute_subpixel_position called with %d %d %f %x %x\n",
             ap_x,ap_y,orientation_in_degrees,pos_x,pos_y);

    if (ap_x < 4 ||
      ap_x > img->width-4 ||
      ap_y < 4 ||
      ap_y > img->height-4){
      return 0;
    }

    /* should we sample horizontally or vertically  */
    direction = calib_orient_from_angle(orientation_in_degrees);

    if (direction == CALIB_HORIZ){
      p0=*(img->rat[ap_y-4]+img->x_offset-ap_x);
      p1=*(img->rat[ap_y-3]+img->x_offset+ap_x);
      p2=*(img->rat[ap_y-2]+img->x_offset-ap_x);
      p3=*(img->rat[ap_y-1]+img->x_offset+ap_x);
      p4=*(img->rat[ap_y]+img->x_offset+ap_x);
      p5=*(img->rat[ap_y+1]+img->x_offset-ap_x);
      p6=*(img->rat[ap_y+2]+img->x_offset+ap_x);
      p7=*(img->rat[ap_y+3]+img->x_offset+ap_x);

      d0=p1-p0;
      d1=p2-p1;
      d2=p3-p2;
      d3=p4-p3;
      d4=p5-p4;
      d5=p6-p5;
      d6=p7-p6;

      d_max = max(d0,max(d1,max(d2,max(d3,max(d4,max(d5,d6))))));
      d_min = min(d0,min(d1,min(d2,min(d3,min(d4,min(d5,d6))))));

      if (-d_min > d_max){
          d0=-d0;
          d1=-d1;
          d2=-d2;
          d3=-d3;
          d4=-d4;
          d5=-d5;
          d6=-d6;
          d_max =-d_min;
      }

      *pos_x = ap_x*1. + 0.5;

      /* If we're at a maximum return the subpixel position */
```

```
    if (d_max > EDGE_THRESHOLD){
        if (d1 == d_max && d3 != d_max){
          *pos_y = ap_y + cz_parfit(d0,d1,d2)/65536.-2.;
          return 1;
        } else if (d2 == d_max && d4 != d_max){
          *pos_y = ap_y + cz_parfit(d1,d2,d3)/65536.-1.;
          return 1;
        } else if (d3 == d_max && d5 != d_max){
          *pos_y = ap_y + cz_parfit(d2,d3,d4)/65536.;
          return 1;
        } else if (d4 == d_max && d6 != d_max){
          *pos_y = ap_y + cz_parfit(d3,d4,d5)/65536.+1.;
          return 1;
        } else if (d5 == d_max){
          *pos_y = ap_y + cz_parfit(d4,d5,d6)/65536.+2.;
          return 1;
        }
        /* If we're not at a maximum, return 0 (something's wrong
         */
        return 0;
    }
  } else {
    p0=*(img->rat[ap_y]+img->x_offset+ap_x-4);
    p1=*(img->rat[ap_y]+img->x_offset+ap_x-3);
    p2=*(img->rat[ap_y]+img->x_offset+ap_x-2);
    p3=*(img->rat[ap_y]+img->x_offset+ap_x-1);
    p4=*(img->rat[ap_y]+img->x_offset+ap_x);
    p5=*(img->rat[ap_y]+img->x_offset+ap_x+1);
    p6=*(img->rat[ap_y]+img->x_offset+ap_x+2);
    p7=*(img->rat[ap_y]+img->x_offset+ap_x+3);

    d0=p1-p0;
    d1=p2-p1;
    d2=p3-p2;
    d3=p4-p3;
    d4=p5-p4;
    d5=p6-p5;
    d6=p7-p6;

    d_max = max(d0,max(d1,max(d2,max(d3,max(d4,max(d5,d6))))));
    d_min = min(d0,min(d1,min(d2,min(d3,min(d4,min(d5,d6))))));

    if (-d_min > d_max){
        d0=-d0;
        d1=-d1;
        d2=-d2;
        d3=-d3;
        d4=-d4;
        d5=-d5;
        d6=-d6;
        d_max =-d_min;
    }

    /* If we're at a maximum, compute the subpixel position and
```

```
         * return 1
         */

        *pos_y = ap_y*1. + 0.5;

        if (d_max > EDGE_THRESHOLD){
            if (d1 == d_max && d3 != d_max){
              *pos_x = ap_x*1. + cz_parfit(d0,d1,d2)/65536.-1.;
              return 1;
            }
            else if (d2 == d_max && d4 != d_max){
              *pos_x = ap_x*1. + cz_parfit(d1,d2,d3)/65536.-1.;
              return 1;
            }
            else if (d3 == d_max && d5 != d_max){
              *pos_x = ap_x*1. + cz_parfit(d2,d3,d4)/65536.;
              return 1;
            }
            else if (d4 == d_max && d6 != d_max){
              *pos_x = ap_x*1. + cz_parfit(d3,d4,d5)/65536.-1.;
              return 1;
            }
            else if (d5 == d_max){
              *pos_x = ap_x*1. + cz_parfit(d4,d5,d6)/65536.-2.;
              return 1;
            }
            /* If we're not at a maximum, something's wrong, and return 0
             */

            return 0;
        }
    }
}


/* enumerate_points_and_fit_line_along_ray() enumerates edge points
 * along a ray from a start_point to an end_point by using calipers
 * (and possibly subpixel interpolation) to compute edge positions
 * enumerate_points_and_fit_line_along_ray() continues enumerating
 * points until one of the following conditions occurs:

 * 0. we reach the end_point
 * 1. no edge is found by the caliper
 * 2. more than one edge is found by the caliper and the highest
 *    scoring edge is less than twice the score of the second highest
 *    scoring edge (this 2X comes from the CONFUSION_THRESHOLD)
 * 3. the computed edge point is an outlier (the distance between the
 *    edge point and the nominal line is larger than threshold)
 * 4. we run off the cip_buffer (screen)

 * After it has enumerated the edge points, it then calls
 * fit_points_to_line to compute the optimal line estimate

 * img - cip_buffer containing a crosshair target
```

```
 * start_point,end_point - define region along which to apply calipers
 * nominal_line, threshold - used to determine whether an edge point
 *                           is an outlier (and consequently end the
 *                       enumeration of edge points). The outlier
 *                       test involves checking whether the
 *                       distance from the point to the
 * step_size - frequency of edge point sampling. Since we want to
 *             evenly sample the lines, we move step_size number of
 *           pels in the larger direction (and a fractional number
 *           of pels in the other direction). This approach provides
 *           even sampling assuming we are doing on-axes calipering
 *           or parabolic subpixel interpolation
 * search_caliper - caliper used to search for edge points
 * computed_line - cwa_line where result will be returned
 * enumerated_points - optional array of cwa_points where found edge
 *                     points would be stored
 * num_points - pointer to an int where the number of found points
 *              would be stored
 * do_subpixel_interp - flag characterizing whether we want to perform
 *                      subpixel interpolation after applying the caliper
 */

static int enumerate_points_and_fit_line_along_ray
(cip_buffer *img, cwa_point *start_point, cwa_point *end_point,
 cwa_line *nominal_line, double threshold, double step_size,
 cclp_caliper *search_caliper, cwa_line *computed_line, cwa_point
 *enumerated_points, int *num_points, int do_subpixel_interp
{
    int i,j,k;
    double maxAbsCosSin;
    cwa_point ap_point, *enum_point, vec;
    int ap_x, ap_y;
    cclp_results res_h[N_EDGES*2-1];
    cclp_params cpp = {EDGE_THRESHOLD*10, N_EDGES, 0};
    double cclp_angle;
    int num_samples,tmp;
    double caliper_vert;
    cct_signal sig;
    cwa_line sampling_line;

    if (cwa_find_target_debug_flag & 0x1){
      printf("enumerated_points_and_fit_line_along_ray called     with\n");
      printf("img %x start_point %f %f end_point %f %f\n",
             img,start_point->x,start_point->y,end_point->x,end_point->y);
      printf("nominal_line %f %f t %f threshold %f\n",
             nominal_line->x,nominal_line->y,nominal_line->t,threshold);
    }

    /* Compute the sampling line */
    sampling_line.x = start_point->x;
    sampling_line.y = start_point->y;
    sampling_line.t = atan2(end_point->y-start_point->y,
                      end_point->x-start_point->x);
```

```
    /* Compute the step which is integral in x or y coordinates
     */

    if (fabs(cos(sampling_line.t)) > fabs(sin(sampling_line.t:
      maxAbsCosSin = fabs(cos(sampling_line.t));
    else
      maxAbsCosSin = fabs(sin(sampling_line.t));

    vec.x = (cos(sampling_line.t)/maxAbsCosSin ) * step_size;
    vec.y = (sin(sampling_line.t)/maxAbsCosSin) * step_size;

    /* Compute the number of samples which we will be making
     */

    num_samples =
      max(abs((int)(start_point->x - end_point->x)),
          abs((int)(start_point->y - end_point->y)),
      step_size;

    if (!enumerated_points)
      enumerated_points =
          (cwa_point *)chp_alloca(num_samples*sizeof(cwa_point

    if (!num_points)
      num_points = &tmp;

    cclp_angle = sampling_line.t*180./PI-90.;

    if (do_subpixel_interp){
      if (calib_orient_from_angle(cclp_angle) == CALIB_VERT)
          cclp_angle = 0;
      else cclp_angle = 90;
    }

    if (cwa_find_target_debug_flag & 0x1)
      printf("cclp_angle %f\n",cclp_angle);


/* Inner Loop: Enumerate edge points
 */

    for (i = 0, ap_point.x = start_point->x, ap_point.y =
           start_point->y, enum_point = enumerated_points,
           *num_points = 0;
         i < num_samples;
         i++, ap_point.x += vec.x, ap_point.y += vec.y, enum_point++,
           (*num_points)++){

      ap_x = (int)(ap_point.x + 0.5);
      ap_y = (int)(ap_point.y + 0.5);

      if (cwa_find_target_debug_flag & 0x1)
          printf("ap_x %d ap_y %d\n",ap_x,ap_y);
```

```
  res_h[0].found = res_h[1].found = 0;


  /* cip_transform throws CGEN_ERR_BADARG, CIP_ERR_PELADDR,
   * CCLP_ERR_FIT when caliper region extends past the bounds of
   * the cip_buffer. We want to break off the search at this
   * point anyways, so we catch the error and jump to fitting a
   * line to the points
   */

  if (sig = cct_catch(CGEN_ERR_BADARG))
      goto fit_line;

  if (sig = cct_catch(CIP_ERR_PELADDR))
      goto fit_line;

  if (sig = cct_catch(CCLP_ERR_FIT))
      goto fit_line;

  cclp_apply
      (search_caliper, img.ap_x, ap_y, cclp_angle, &npp, res_h);

  cct_end(CCLP_ERR_FIT);
  cct_end(CIP_ERR_PELADDR);
  cct_end(CGEN_ERR_BADARG);

  /* check if edge is not found */
  if (!res_h[0].found)
      goto fit_line;

  /* check if more than one edge is found CONFUSION */
  if (res_h[1].found &&
      res_h[0].score < CONFUSION_THRESHOLD*res_h[1].score) {
      if (cwa_find_target_debug_flag & 0x1) {
        cd_showas(res_h,"cclp_results");
        cd_showas(&res_h[1],"cclp_results");
      }
      goto fit_line;
  }

  if (cwa_find_target_debug_flag & 0x1){
      printf("ap_x %d ap_y %d angle %f pos %f\n",
           ap_x,ap_y,cclp_angle,res_h[0].position);
      if (fabs(res_h[0].position) > 1.){
        cd_showas(res_h,"cclp_results");
        cd_showas(&res_h[1],"cclp_results");
      }
  }

  /* compute the subpixel point measured by the caliper */

  enum_point->x =
      ap_x + 0.5 + res_h[0].position*cos(cclp_angle*PI/180);
  enum_point->y =
```

```
            ap_y = 0.5 * res_h[0].position*sin(cclp_angle*PI/180);
    #if 0
          if (do_subpixel_interp){

              if (!compute_subpixel_position
                (img, (int) enum_point->x, (int) enum_point->y,
                 cclp_angle, &enum_point->x,&enum_point->y)
                goto fit_line;
              if (cwa_find_target_debug_flag & 0x1)
                printf("compute subpixel position returned %f %f\n",
                       enum_point->x,enum_point->y);
          }
    #endif
          /* check if the point is within the ACCEPTABLE REGION
           * (within distance threshold of the nominal_line
           */

          if (distance_from_point_to_line(enum_point, nominal_line) >
              threshold)
              goto fit_line;
        }

    fit_line:
          /* fit a line to the points and return 0 if we encountered
           * any bad points, return 1 otherwise
           */

        fit_line_to_points
          (enumerated_points, *num_points, computed_line);
    }

    static int round_number(double x)
    {
        if (x > 0)
          return (int)(x+0.5);
        else
          return(int)(x-0.5);
    }

    /* average two orientations but take into account that orientations
     * mod PI are equivalent
     */
    static double average_two_orientations(double orient_1, double orient_2)
    {
        double diff,res;
        int modDiff;

        if (cwa_find_target_debug_flag & 0x1)
          printf("avg %f %f ",orient_1,orient_2);

        diff = orient_2 - orient_1;
        modDiff = round_number(diff/PI);
        res= ((orient_1+orient_2-modDiff*PI)/2);
```

```
    if (cwa_find_target_debug_flag & 0x1)
      printf(" = %f\n",res);
    return res;
}

/* compute angle in alternative coordinates given by the map
 */
double transform_angle_according_to_calibration_map
(priv_transform *map, double img_orientation)
{
    double realSin,realCos;

    cz_transformPoint(map,cos(img_orientation),sin(img_orientation),
                      &realCos,&realSin);
    return(atan2(realSin,realCos));
}
/* average the orientations of the four rays to compute the best
 * composite orientation estimate. Given the orientations of the four
 * image rays (the rays should be oriented in multiples of PI/2 out
 * with each other), compute the "average" orientation. First, the
 * orientations are aligned with img_orient[0]. Since if img_orient[0]
 * shifted by PI/2 to become horizontal, then all of the angles may be
 * orientations.

 * This function transforms the image
 * orientations into the physical domain in order to exploit the
 * constraint that the physical lines are orthogonal
 */

static double average_four_image_orientations
(cwa_calib *calib, double *img_orient)
{
    int i;
    double phys_orient[4], avg_phys_orient,avg_img_orient;

    /* First, transform angles from image coordiantes to physical
     * coordinates
     */
    for (i = 0; i < 4; i++)
      phys_orient[i] =
          transform_angle_according_to_calibration_map
          (((priv_cwa_calib *)calib)->map,img_orient[i]);

    if (cwa_find_target_debug_flag & 0x1)
      for (i = 1; i < 4; i++)
          printf("%f\n",phys_orient[i]);

    /* Shift orthogonal angles by PI/2 in order to line up with the
     * first orientation
     */
    for (i = 1; i < 4; i++)
      if (fabs(sin(phys_orient[i]-phys_orient[0]))>sqrt(2.5))
          phys_orient[i] += PI/2;
```

```
    /* average the orientations */
    avg_phys_orient =
      average_two_orientations
      (average_two_orientations(phys_orient[0],phys_orient[1]),
       average_two_orientations(phys_orient[2],phys_orient[3]));

    /* compute the orientation in image coordinates corresponding to
     * the averaged orientation
     */
    avg_img_orient =
      transform_angle_according_to_calibration_map
      (((priv_cwa_calib *)calib)->inv_map,avg_phys_orient);

    if (cwa_find_target_debug_flag & 0x1)
      printf("phys %f img %f\n",avg_phys_orient,avg_img_orient);

    return avg_img_orient;

}

/* compute the image orientation for which the physical orientation
 * is orthogonal to another image orientation
 */
static double image_normal_to_image_angle
(cwa_calib *calib, double img_orient)
{

    return
      (transform_angle_according_to_calibration_map
       (((priv_cwa_calib *)calib)->inv_map,
        (transform_angle_according_to_calibration_map
         (((priv_cwa_calib *)calib)->map,img_orient) + PI/2)));
}

/*
 * compute_center_position_and_angle_assuming_four_guaranteed_good_rays()
 * improve estimate of the crosshair position and orientation by
 * searching four regions of guaranteed good length (50 pixels)
 * eminating from the crosshair center. Lines are fit to the edge
 * points along the four regions, and then the orientations of these
 * lines are "averaged". Two orthogonal lines are computed by
 * averaging pairs of the four rays Finally, the position estimate is
 * computed by intersecting the two orthogonal lines

 * img - cip_buffer containing an image of a crosshair target
 * calib - cwa_calib object characterizing pixel non-rectangularity
 * estimated_ctr_img - estimated position (given in image
 *                     coordinates) of crosshair target (computed in step 1)
 * estimated_orientation_phys - user-supplied orientation estimate
 *                              (given in physical coordinates)
 * guaranteed_good_distance - length (in pels) of guarateed good
 *                            region extending outward from the origin
 * computed_center - structure where computed position will be stored
```

```
 * computed_orientation_phys - double where computed orientation  in
 *                             physical coordinates) will be stored
 * good_rays - array of four cwa_line structures where good rays
 *             (lines fit to sampled edge points in the good region)
 *           will be stored
 */

static int compute_center_position_and_angle_assuming_four_guaranteed_good_rays
(cip_buffer *img, cwa_calib *calib, cwa_point *estimated_ctr_img, double
 estimated_orientation_phys, double num_guaranteed_good_points,
 cwa_point *computed_center, double *computed_orientation_phys,
 cwa_line *good_rays)
{
    int i;
    double angle[4];
    cwa_point start_point[4],end_point[4];
    cwa_line computed_line[4], nominal_line;
    double img_angle[4],angle_horiz_line;

    angle[0] = estimated_orientation_phys;
    angle[1] = estimated_orientation_phys+PI/2;
    angle[2] = estimated_orientation_phys+PI;
    angle[3] = estimated_orientation_phys+PI*1.5;

    /* We define the 50 pel GUARANTEED GOOD REGION in terms of a
     * VALID REGION so that we can use
     * the enumerate_points_and_fit_line_along_ray() function
     * If we're sampling along a line and we want to stop sampling
     * after 50 pels, then we can use a nominal line which is normal
     * to and passes through the start point, and a threshold of 50
     */
    nominal_line.x = estimated_ctr_img->x;
    nominal_line.y = estimated_ctr_img->y;

    /* The passed orientation is given in physical coordinates */

    for (i = 0; i < 4; i++)
      angle[i] = transform_angle_according_to_calibration_map
          (((priv_cwa_calib *)calib)->inv_map,angle[i]);

    for (i = 0; i < 4; i++){

      /* The sampling region begins at the point a small distance
       * away from the crosshair center
       * (CIRCULAR_BUFFER_ZONE_AROUND_CENTER) to be exact, and
       * extends out to guaranteed_good_distance from the center
       */
      start_point[i].x = estimated_ctr_img->x +
          CIRCULAR_BUFFER_ZONE_AROUND_CENTER*cos(angle[i]);
      start_point[i].y = estimated_ctr_img->y +
          CIRCULAR_BUFFER_ZONE_AROUND_CENTER*sin(angle[i]);
      end_point[i].x = estimated_ctr_img->x +
          num_guaranteed_good_points*cos(angle[i]);
      end_point[i].y = estimated_ctr_img->y +
```

```
        num_guaranteed_good_points*sin(angle[i]);
    nominal_line.t = angle[i]+PI/2;

    enumerate_points_and_fit_line_along_ray
        (img,&start_point[i],&end_point[i],&nominal_line,
         num_guaranteed_good_points,
         STEP_SIZE_USED_TO_ENUMERATE_POINTS_IN_GOOD_REGION*2.,
         &clp_find_target_initial_search,
         &computed_line[i],NULL,NULL,0);
    img_angle[i] = computed_line[i].t;

    cu_copy(&computed_line[i],&good_rays[i],sizeof(cwa_line));
  }

  /* average the angles */

  angle_horiz_line =
    average_four_image_orientations(calib,img_angle);

  if (cwa_find_target_debug_flag & 0x1)
    printf("average %f %f %f %f %f\n",
           img_angle[0],img_angle[1],img_angle[2],img_angle[3],
           angle_horiz_line);

  /* average the fit rays */

  computed_line[0].x = (computed_line[0].x + computed_line[2].x)/2;
  computed_line[0].y = (computed_line[0].y + computed_line[2].y)/2;
  computed_line[1].x = (computed_line[1].x + computed_line[3].x)/2;
  computed_line[1].y = (computed_line[1].y + computed_line[3].y)/2;
  computed_line[0].t = angle_horiz_line;
  computed_line[1].t = image_normal_to_image_angle(calib,angle_horiz_line);

  /* compute the intersected point */
  cwa_lines_to_point(&computed_line[0],&computed_line[1],computed_center);

  /* fix the orientations of the good rays to correspond to the
   * "average" orientation
   */
  good_rays[0].t = good_rays[2].t = angle_horiz_line;

  *computed_orientation_phys =
    transform_angle_according_to_calibration_map
    (((priv_cwa_calib *)calib)->map,angle_horiz_line);
  good_rays[1].t = good_rays[3].t =
    transform_angle_according_to_calibration_map
    (((priv_cwa_calib *)calib)->inv_map,
     *computed_orientation_phys+PI/2);
}

/* Compute the two intersections between a circle and a line if the
 * circle and line intersect. If the circle and line do not
 * intersection, signal CWA_ERR_NC_INTERSECTION
 */
```

```
static int intersect_line_and_circle(cwa_line *line, cwa_point *pt,
                                     double radius, cwa_point *res_1,
                                     cwa_point *res_2)
{
    double angle_to_line, distance_to_line, angle_to_point_on_line;

    angle_to_line = line->t+PI/2;
    if ((cos(angle_to_line)*(line->x-pt->x)+
        sin(angle_to_line)*(line->y-pt->y)) < 0)
      angle_to_line += PI;
    distance_to_line = distance_from_point_to_line(pt,line);

    if (distance_to_line > (radius - EPSILON_THRESHOLD))
      cct_error(CWA_ERR_NO_INTERSECT);

    angle_to_point_on_line = acos(distance_to_line/radius);

    res_1->x = pt->x +
      radius*cos(angle_to_line+angle_to_point_on_line);
    res_1->y = pt->y +
      radius*sin(angle_to_line+angle_to_point_on_line);
    res_2->x = pt->x +
      radius*cos(angle_to_line-angle_to_point_on_line);
    res_2->y = pt->y +
      radius*sin(angle_to_line-angle_to_point_on_line); /
}


/* compute_center_position_and_angle_along_guaranteed_good_rays()
 * generate final estimate of crosshair position and orientation by
 * enumerating points along four rays (using four good rays as
 * thresholding lines). We use tighter calipers because we trust the
 * positions of the good rays.

 * img - cip_buffer containing an image of a crosshair target
 * calib - cwa_calib object characterizing pixel non-rectangularity
 * estimated_ctr_img - estimated position (given in image
 *                     coordinates) of crosshair target (computed in
 *                 step 2)
 * good_rays - array of four cwa_line structures where good rays
 *             (lines fit to sampled edge points in the good region)
 *           will be stored
 * tight_threshold - threshold used for determining outlier points
 *                   with respect to the good rays
 * computed_center - structure where computed position will be stored
 * computed_orientation_phys - double where computed orientation (in
 *                             physical coordinates) will be stored
*/

static int compute_center_position_and_angle_along_guaranteed_good_rays
(cip_buffer *img, cwa_calib *calib, cwa_point *estimated_ctr_img,
 cwa_line *good_rays, double tight_threshold, cwa_point
 *computed_center, double *computed_orientation_phys,
 double *feature_score)
```

```
{
    int i;
    double angle[4];
    cwa_point start_point[4],end_point[4];
    cwa_line computed_line[4];
    double img_angle[4],angle_horiz_line;
    int num_enumerated_points[4];
    cwa_point possible_start_points[2], enumerated_points[4][MAX_NUM_POINTS];
    double avg_squared_error;

    if (cwa_find_target_debug_flag & 0x1)
      printf("compute_center_position_and_angle_along_guaranteed_good_rays\n");

    for (i = 0; i < 4; i++){
      double distance_to_center,radius,orientation_away_from_center;

          /* For each ray, we need to determine what the sampling
           * line should be
           * We do this by intersecting the ray with a circle
           * centered at the crosshair center
           * Since there are two intersection points, we need to
           * determine which one is the real start of the sampling
           * line
           * It turns out that we can simply pick the intersection
           * point which is closer to the center of the ray
           */

      distance_to_center =
          distance_from_point_to_line(estimated_ctr_img,&good_rays[i]);

      radius = sqrt(CIRCULAR_BUFFER_ZONE_AROUND_CENTER*
                  CIRCULAR_BUFFER_ZONE_AROUND_CENTER-
                  distance_to_center*distance_to_center);

      intersect_line_and_circle(&good_rays[i],estimated_ctr_img,
                          radius,&possible_start_points[0],
                          &possible_start_points[1]);
      if (cwa_point_distance
          (&possible_start_points[0],(cwa_point *)&good_rays[i])<
          cwa_point_distance
          (&possible_start_points[1],(cwa_point *)&good_rays[i]))
          cu_copy(&possible_start_points[0],&start_point[i],
                sizeof(cwa_point));
      else
          cu_copy(&possible_start_points[1],&start_point[i],
                sizeof(cwa_point));

      /* Now, we have to determine which way we should travel along
       * the ray (good_ray[i].t or good_ray[i].t+PI
       * We determine which way by comparing the start point with
       * the center of the good ray
       */

      orientation_away_from_center = good_rays[i].t;
```

```
  if ((cos(orientation_away_from_center)*
       (good_rays[i].x-estimated_ctr_img->x, -
       sin(orientation_away_from_center)*
       (good_rays[i].y-estimated_ctr_img->y) < 0)
      orientation_away_from_center += PI;

  end_point[i].x = start_point[i].x +
      MAX_LENGTH_DIAGONAL_ACROSS_CIP_BUFFER*
      cos(orientation_away_from_center);
  end_point[i].y = start_point[i].y +
      MAX_LENGTH_DIAGONAL_ACROSS_CIP_BUFFER*
      sin(orientation_away_from_center);

  if (cwa_find_target_debug_flag & 0x1){
      printf("center %f %f good_ray %f %f %f distance %f\n",
           estimated_ctr_img->x,estimated_ctr_img->y,
           good_rays[i].x,good_rays[i].y,good_rays[i].t,
           distance_to_center);
      printf("%f %f %f %f\n",
           start_point[i].x,start_point[i].y,
           end_point[i].x,end_point[i].y);
  }


  enumerate_points_and_fit_line_along_ray
      (img,&start_point[i],&end_point[i],&good_rays[i],
       tight_threshold,
       STEP_SIZE_USED_TO_ENUMERATE_ALL_POINTS,
       &clp_find_target_fine_search,
       /*&clp_find_target_compute_subpixel,*/
       &computed_line[i],&enumerated_points[i][0],
       &num_enumerated_points[i],1);
  img_angle[i] = computed_line[i].t;
}

/* For each pair of opposite rays, use the closest points so that
 * we use the same number of points on both sides of the
 * crosshair center
 * Since we don't know why the last edge point became invalid, we
 * skip the neighboring 2 points just to be on the safe side
 */

num_enumerated_points[0] = num_enumerated_points[2] =
  min(num_enumerated_points[0],num_enumerated_points[2])-OUTLIER_BUFFER;
num_enumerated_points[1] = num_enumerated_points[3] =
  min(num_enumerated_points[1],num_enumerated_points[3])-OUTLIER_BUFFER;

/* Re-fit lines to these points */

avg_squared_error = 0;

for (i = 0; i < 4; i++){
  double fit_error;
```

```
      fit_error =
          fit_line_to_points
          (&enumerated_points[i][0], num_enumerated_points[i],
           &computed_line[i]);
      avg_squared_error += fit_error/4;
      img_angle[i] = computed_line[i].t;
    }

    *feature_score =
      cz_map_average_squared_error_to_max_1000(avg_squared_error);

    /* Average the four rays by averaging the positions and
     * orientations
     */

    computed_line[0].x = (computed_line[0].x + computed_line[2].x)/2;
    computed_line[0].y = (computed_line[0].y + computed_line[2].y)/2;
    computed_line[1].x = (computed_line[1].x + computed_line[3].x)/2;
    computed_line[1].y = (computed_line[1].y + computed_line[3].y)/2;

    angle_horiz_line =
      average_four_image_orientations(calib,img_angle
    computed_line[0].t = angle_horiz_line;
    computed_line[1].t = image_normal_to_image_angle(calib,angle_horiz_line);


    cwa_lines_to_point(&computed_line[0],&computed_line[1],computed_center);

    *computed_orientation_phys =
      transform_angle_according_to_calibration_map
      (((priv_cwa_calib *)calib)->map,angle_horiz_line);
}

/* cwa_find_target() is the top level function for localizing
 * the crosshair target. It returns 0 if it cannot find the crosshair.
 * 1) use search to improve position estimate of crosshair center
 * 2) compute four good rays eminating from the origin by sampling
 *    edge points within the guaranteed good region along the user
 *      specified orientation
 * 3) Enumerate subpixel edge points along the four rays eminating
 *    from the the origin and use the found good rays to identify outliers

 * cwa_find_target tries to localize a crosshair target within an
 * image and returns 1 if it was successful and 0 otherwise

 * img - cip_buffer containing crosshair target
 * calib - characterizing field of view
 * angle - initial orientation estimate of crosshair target
 * result - pointer to cwa_line structure where result will be stored
 */
int cwa_find_target_internal
(cip_buffer *img, cwa_calib *calib, double angle, cwa_line *result,
 cwa_status *status)
```

```
{
    cwa_point gross_pos_estimate,improved_pos_estimate;
    double improved_phys_orientation_estimate;
    cwa_line good_rays[4];
    int i,ans;
    cct_signal sig;
    ctm_timer time;
    int t0,t1,t2,t3;
    cwa_line tmp;
    cwa_calib *calib_identity=NULL;

    NO_REGISTER(calib_identity);

    cu_clear(status,sizeof(cwa_status));

    /* CWA_ERR_NO_INTERSECTION is used to signify when we cannot find
     * any edge points for an expected horizontal or vertical line
     */

    if (sig = cct_catch(0))
      goto done;

    /* expand the stack to make room for the 4000 double arrays we
     * allocate
     */

    cu_expand_stack(16);

    if (!result)
      result = &tmp;

    if (!img)
      cct_error(CGEN_ERR_BADARG);

    /* If no calibration object is passed, make one so that there is
     * only one path through the code
     */
    if (!calib)
      calib = calib_identity = cwa_calib_make(0.,0.,0.,1.,1.,0);

    /* 1) use search */

    ctm_begin(&time);

    ans = gross_locate_crosshair_center_in_image
      (img,(int)(10. * angle*180/PI), &gross_pos_estimate.);
    if (!ans) cct_error(CWA_ERR_NO_INTERSECT);

    {
      cip_buffer win;
      double x_edge, y_edge;
      int edge_val,dir;
      double angle_offset;
      cct_signal sig;
```

```
  int i;
  double left[4]={-2.,-2.,2.,2.};
  double top[4]={-2.,2.,2.,-2.};
  cip_window(img,&win,
             gross_pos_estimate.x-35,gross_pos_estimate.y-35,70,70);
  for (i = 0; i < 4; i++){
      if (cz_cebt_start_return_success_val
        (&win, 3,3,left[i],top[i],&x_edge, &y_edge, &edge_val,
         &dir, &angle_offset))
        break;
  }
  status->contrast_score=cz_map_edge_val_to_max_1000(edge_val);
}

t0 = ctm_read(&time);

if (cwa_find_target_debug_flag & 0x1){
  printf("gross pos:");
  cd_showas((int)&gross_pos_estimate,"cwa_point");
}

/* 2) improve the position and orientation estimate and compute
 * the four good rays
 */

compute_center_position_and_angle_assuming_four_guaranteed_good_rays
  (img, calib, &gross_pos_estimate,
   transform_angle_according_to_calibration_map
   (((priv_cwa_calib *)calib)->map.angle),
   (double)GUARANTEED_GOOD_SIZE,&improved_pos_estimate,
   &improved_phys_orientation_estimate,good_rays);


t1 = ctm_read(&time);

if (cwa_find_target_debug_flag & 0x1){
  printf("improved_pos_estimate:");
  cd_showas((int)&improved_pos_estimate,"cwa_point");
}

/* 3) Enumerate edge points along those four good rays and use
 * those lines to compute the final position and orientation estimates
 */

compute_center_position_and_angle_along_guaranteed_good_rays
  (img, calib, &improved_pos_estimate,good_rays,TIGHT_THRESHOLD,
   (cwa_point *)result,&(result->t),
   &status->feature_score);

t2 = ctm_read(&time);

cz_transformPoint
  (((priv_cwa_calib *)calib)->map,
   result->x,result->y,
```

```
        &result->x,&result->y);

    if (cwa_find_target_debug_flag & 0x1){
      printf("result:");
      cd_showas((int)result,"cwa_point");
    }

    if (cwa_find_target_debug_flag & 0x4){
      printf("Times:");
      printf("  gross locate: %d\n",t0);
      printf("  fine locate:  %d\n",t1-t0);
      printf("  finer locate: %d\n",t2-t1);
    }

done:
    cu_expand_stack(-16);
    if (calib_identity) cwa_calib_delete(calib_identity);
    /* Make returned angle agree with given angle argument */

    if (sig) cct_throw(sig);

    for (i = -8; i <= 8; i++)
      if ((i != 0) &&
          (fabs(result->t + i*PI/2 - angle) < fabs(result->t - angle))
          result->t += i*PI/2 , i--;

    return 1;
}

int cwa_find_target
(cip_buffer *img, cwa_calib *calib, double angle, cwa_line *result,
 cwa_status *status)
{
    cwa_status tmp_status;
    cct_signal sig;

    if (!status) status = &tmp_status;

    cct_catch(CGEN_ERR_BADARG);
    cct_catch(CWA_ERR_BACKLIT);
    cct_catch(CWA_ERR_TRACKER);

    if (sig=cct_catch(0)){
      if (sig == CGEN_ERR_BADARG){
          status->error_flags |= CWA_ERR_BADARG;
          cct_end(sig);
          cct_error(sig);
      }
      status->error_flags |= CWA_ERR_FAILED_TO_FIND_TARGET;
      return status->found = 0;
    }

    cwa_find_target_internal(img,calib,angle,result,status);
```

```
    return status->found = 1;
}
```

Described herein are calibration targets and methods for determining the location and orientation thereof meeting the objects set forth above. It will be appreciated that the embodiments shown in the drawings and discussed above are merely illustrative of the invention, and that other embodiments incorporating changes therein fall within the scope of the invention, of which we claim:

1. A machine vision method of determining a location of a reference point in an image of a calibration target, the method comprising the steps of

A. generating an image of a target that includes

i. two or more regions, each region being defined by at least two linear edges that are directed toward a reference point;

ii.. at least one of the regions having a different imageable characteristic from an adjacent region;

B. identifying edges in the image corresponding to lines in the calibration target;

C. determining a location where lines fitted to those edges intersect.

2. A method according to step 1, wherein step (B) comprises the step of identifying edges in the image by applying an edge detection vision tool to the image.

3. A method according to step 1, wherein step (B) comprises identifying edges in the image by applying a caliper vision tool to the image.

4. A method according to step 1, wherein step (B) comprises identifying edges in the image by applying a first difference operator vision tool to the image, and by applying a peak detector vision tool the an output of the first difference operator vision tool

5. A method according to step 1, wherein step (B) comprises identifying edges in the image by applying a caliper tool to the image beginning at an approximate location of the reference point.

6. A method according to claim 5, comprising the step of determining an approximate location of the reference point by applying a Hough line vision tool to the image and finding an intersection of lines identified thereby.

7. A method according to claim 5, comprising the step of determining an approximate location of the reference point by applying a correlation vision tool to the image using a template substantially approximating an expected pattern of edges in the image.

8. A method according to claim 5, comprising the step of determining an approximate location of the reference point by applying a projection vision tool to the image along one or more axes with which the edges are expected to be aligned.

9. A method according to claim 1, comprising the steps of

identifying edges in the image corresponding to lines in the calibration target; and

determining an orientation of the target based on the angle of those identified edges.

10. A machine vision according to claim 1, wherein

step (B) includes the step of applying a Hough line vision tool to identify edges in the image corresponding to lines in the calibration target; and

the method further including the step of determining an orientation of the target based on the angle of those identified edges.

11. A machine vision according to claim 1, comprising

A. applying a Sobel edge tool to the image to generate at least a Sobel angle image; and

B. taking an angle histogram of the Sobel angle image to determine an orientation of the target.

12. A machine vision method of an orientation of a calibration target in an image, the method comprising the steps of

A. generating an image of a target that includes

i. two or more regions, each region being defined by at least two linear edges that are directed toward a position;

ii.. at least one of the regions having a different imageable characteristic from an adjacent region;

B. identifying edges in the image corresponding to lines in the calibration target;

C. determining an orientation of the target based on the angle of those identified edges.

13. A method according to claim 12, wherein

step (B) includes applying a Hough line vision tool to the image to identify therein edges in the image corresponding to lines in the calibration target; and

step (C) includes determining an orientation of the target based on the angle of those identified edges.

14. A method according to claim 12, wherein

step (B) includes applying a Sobel edge tool to generate at least a Sobel angle image; and

step (C) includes taking an angle histogram of the Sobel angle image to determine an orientation of the target.

15. A method according to any of claims 1 and 12, wherein step (A) includes the step of generating an image of a target in which each region has any of a different color, contrast, brightness and stippling from regions adjacent thereto.

16. A method according to claim 15, for use with a machine vision system having image capture means for capturing an image of the target, wherein each region has a different characteristic in such an image than the regions adjacent thereto.

17. A method according to any of claims 1 and 12, wherein step (A) includes the step of generating an image of a target for which the reference point is at a center thereof.

18. A method according to claim 17, wherein step (A) includes the step of generating an image of a target for which the linear edges substantially meet at the reference point.

19. A method according to any of claim 1 and 12, wherein step (A) includes the step of generating in image of a target having four imageable regions.

20. A method according to claim 19, wherein step (A) includes the step of generating an image of a four-region target in which the at least two linear edges of each of the regions are perpendicular to one another.

21. A machine vision method of determining a location and orientation of a calibration target in an image thereof, the method comprising the steps of

A. generating an image of a target that includes

i. two or more regions, each region being defined by at least two linear edges that are directed toward a reference point;

ii.. at least one of the regions having a different imageable characteristic from an adjacent region;

B. generating, based on analysis of the image, an estimate of an orientation of the target therein;

C. generating, based on analysis of the image, an estimate of a location of the reference point therein;

D. refining, based on analysis of the image, estimates of at least one of the location of the reference point in the image and the orientation of the target in the image.

22. A method according to claim 21, wherein step (B) includes one or more of the following steps:

i. applying a Hough line vision tool to the image to find an angle of edges in the image;

ii. applying a Sobel edge tool to the image to generate Sobel angle image, generating a histogram of the angles represented in that Sobel angle image, and determining a one dimensional correlation thereof; and

iii. inputting an orientation from the user.

23. A method according to claim 22, wherein step (C) includes one or more of the following steps:

i. applying a Hough vision tool to the image to find an approximate location of edges therein, and finding an intersection of lines defined by those edges;

ii. applying a projection tool vision tool to the image to find an approximate location of edges therein, and finding an intersection of lines defined by those edges;

iii. applying a correlation vision tool to the image to find approximate coordinates of the reference point; and

iv. determining a sum of absolute differences to find approximate coordinates of the reference point.

24. A method according to claim 23, wherein step (D) includes the step of invoking the steps in sequence, at least one time:

i. applying a caliper vision tool along each of the expected edges in the image, where those edges are defined by a current estimate of the reference point location and a current estimate of the target orientation; and

ii. fitting lines to edge points determined by the caliper vision tool.

25. A method according to claim 24, wherein step (D)(i) includes the step of applying the caliper vision tool along each of the expected edges until any of the following conditions is met: no edge is found by the caliper vision tool at a sample point; more than one edge is found by the caliper vision tool at a sample point; a distance between a sample edge point and a nominal line is larger than a threshold; or, the caliper vision tool window for the sample edge point extends outside of the image.

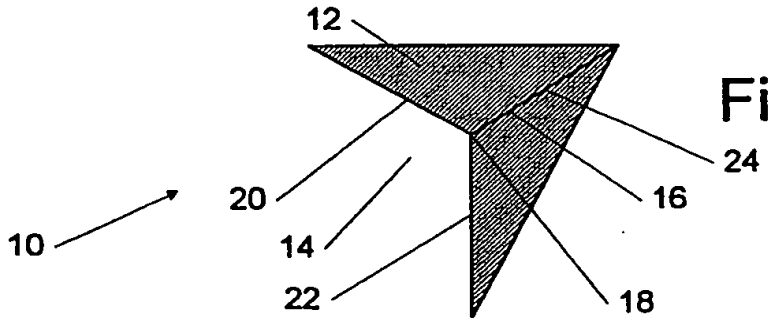


Figure 1A

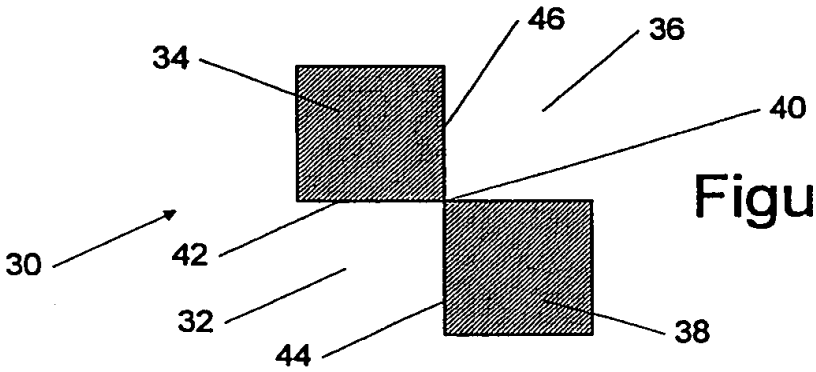


Figure 1B

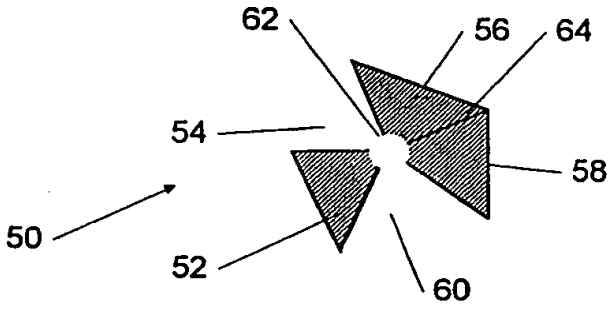


Figure 1C

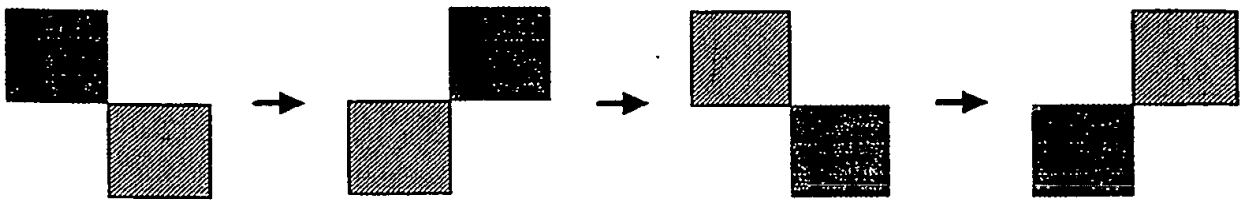


Figure 1D

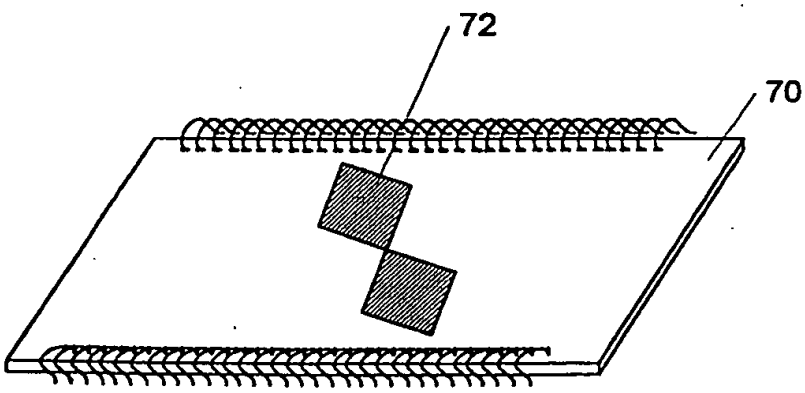


Figure 2

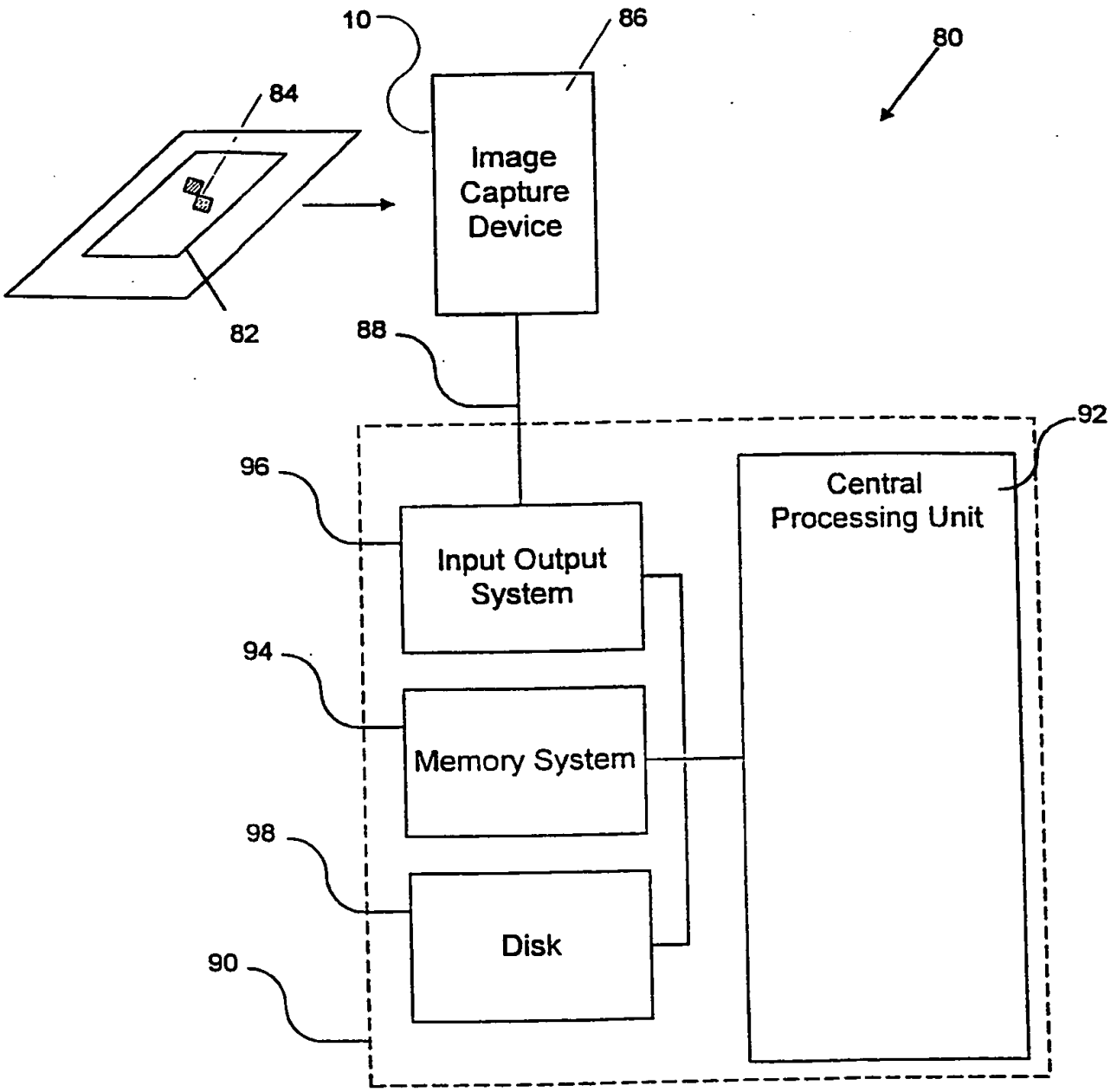

10
86
80
84
Image Capture Device
82
88
92
96
Central Processing Unit
Input Output System
94
Memory System
98
Disk
90

Figure 3

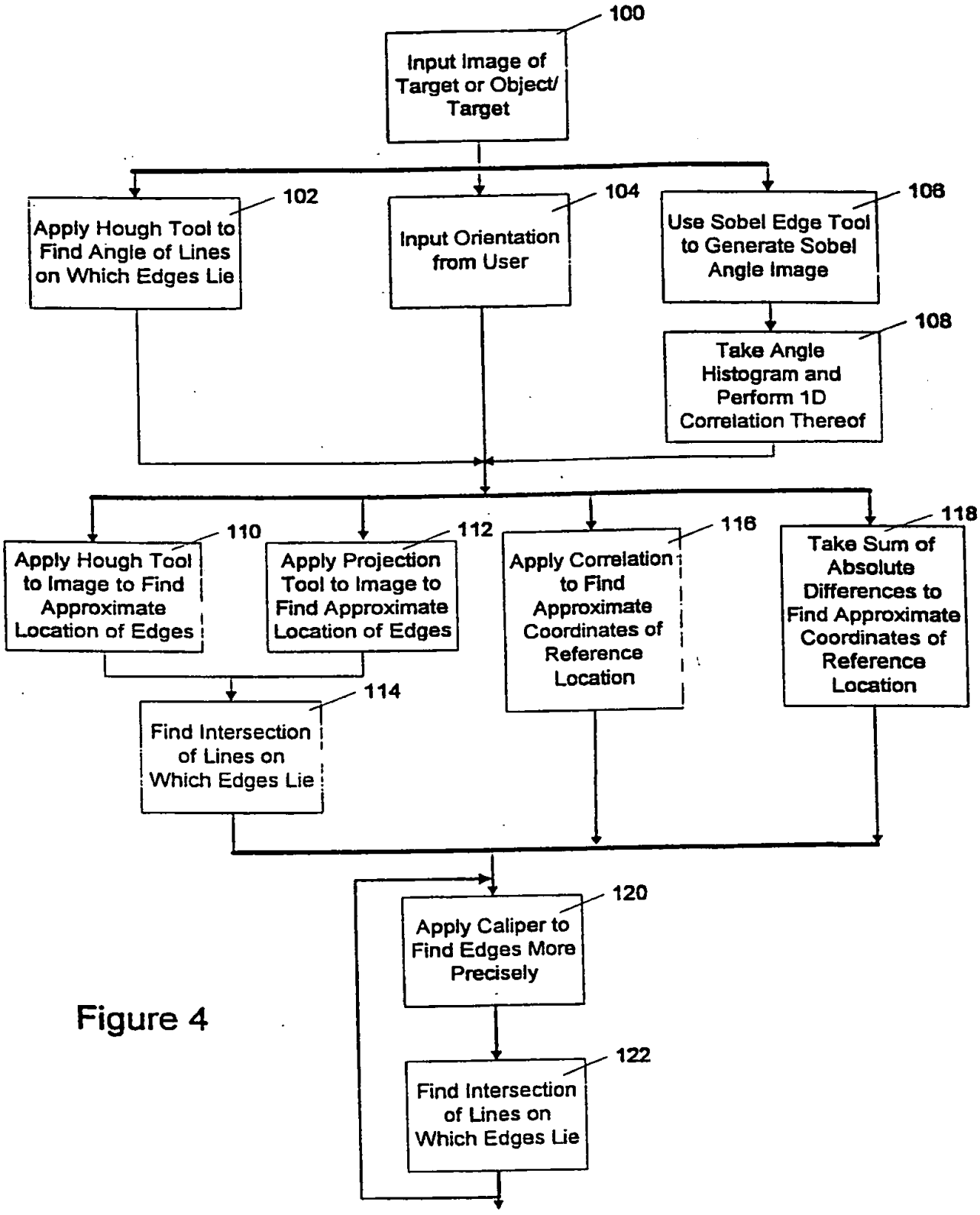

100
Input Image of Target or Object/ Target
102
Apply Hough Tool to Find Angle of Lines on Which Edges Lie
104
Input Orientation from User
106
Use Sobel Edge Tool to Generate Sobel Angle Image
108
Take Angle Histogram and Perform 1D Correlation Thereof
110
Apply Hough Tool to Image to Find Approximate Location of Edges
112
Apply Projection Tool to Image to Find Approximate Location of Edges
114
Find Intersection of Lines on Which Edges Lie
116
Apply Correlation to Find Approximate Coordinates of Reference Location
118
Take Sum of Absolute Differences to Find Approximate Coordinates of Reference Location
120
Apply Caliper to Find Edges More Precisely
122
Find Intersection of Lines on Which Edges Lie

Figure 4

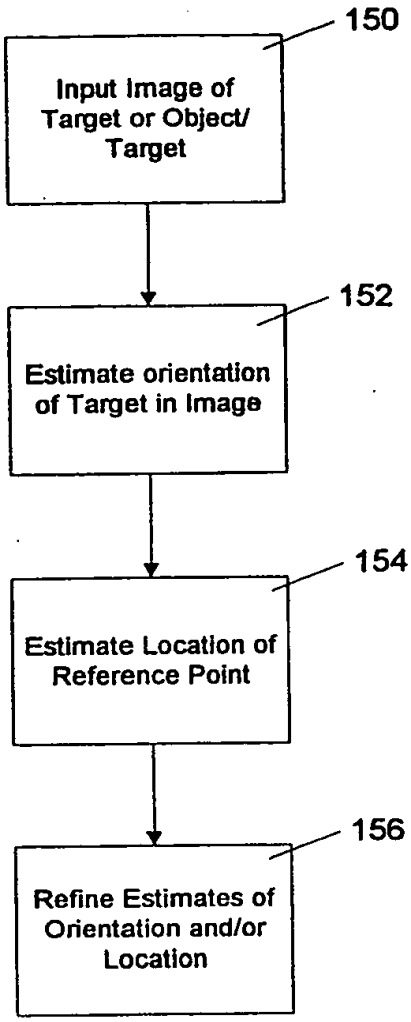

150
Input Image of Target or Object/ Target
152
Estimate orientation of Target in Image
154
Estimate Location of Reference Point
156
Refine Estimates of Orientation and/or Location

Figure 5

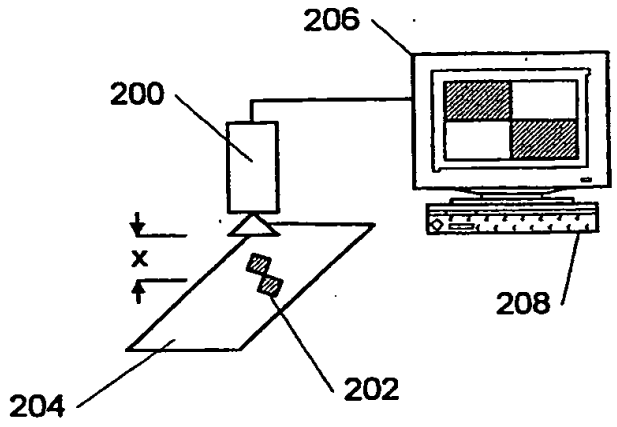


Figure 6A

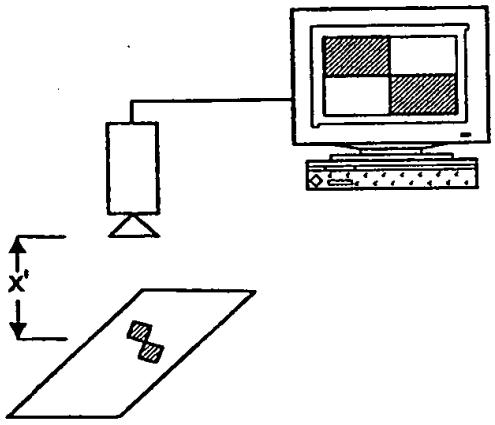


Figure 6B

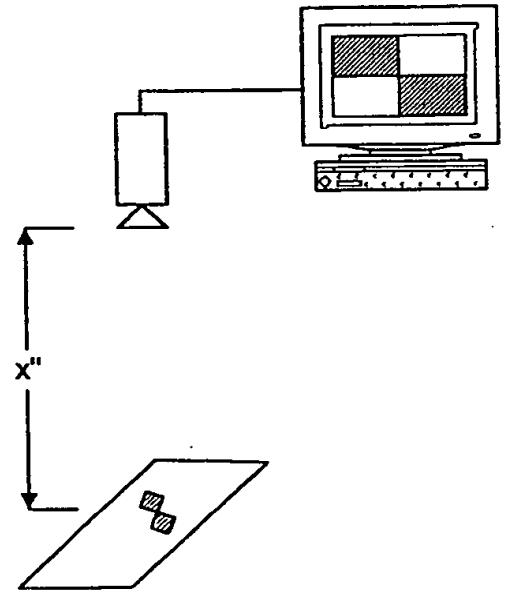


Figure 6C